Microsoft Open Database Connectivity (ODBC)
Microsoft Open Database Connectivity (ODBC)

The Microsoft Open Database Connectivity (ODBC) interface is a C programming language interface that makes it possible for applications to access data from a variety of database management systems (DBMSs). ODBC is a low-level, high-performance interface that is designed specifically for relational data stores.

The ODBC interface allows maximum interoperability—an application can access data in diverse DBMSs through a single interface. Moreover, that application will be independent of any DBMS from which it accesses data. Users of the application can add software components called drivers, which interface between an application and a specific DBMS.

Documentation

- ODBC Programmer's Reference
  Documentation of ODBC interfaces and methods.
- ODBC Data Source Administrator
  The Microsoft ODBC Data Source Administrator manages database drivers and data sources.
- Microsoft-Supplied ODBC Drivers
  Documentation for the ODBC Desktop Database Drivers, the ODBC Driver for Oracle, and the Visual FoxPro® ODBC Driver.
- ODBC Test
  Microsoft ODBC Test is an ODBC-enabled application that you can use to test ODBC drivers.
- ODBC Glossary
  Defines terms relevant to ODBC.
- ODBCValidconf.EXE
  Describes the command-line utility for configuring drivers and data source names.

Support

Information about support options can be found on the Microsoft Help and Support Web site.

© 2016 Microsoft

What's New in ODBC 3.8

Windows 8 includes an updated version of ODBC 3.8. ODBC 3.8 in Windows 8 includes the following features:

- Driver-Aware Connection Pooling
- Asynchronous Execution (Notification Method)
- Data Access Tracing (Windows 8)
- PowerShell commands have been added to help you manage ODBC data sources, ODBC drivers, ODBC Performance Counter, and data access tracing at the command line. For more information see Windows Data Access Components PowerShell Commands.

Windows 7 includes an updated version of ODBC, ODBC 3.8. ODBC 3.8 includes the following features:

- Executing connection operations asynchronously. For more information, see Asynchronous Execution (Polling Method).
- Streamed output parameters. For more information, see Retrieving Output Parameters Using SQLGetData.
- ODBC C data type extensibility. For more information, see C Data Types in ODBC.

Driver writers should read Upgrading a 3.5 Driver to a 3.8 Driver.

Asynchronous connection operations can be used by ODBC 3.x and ODBC 2.x applications with an ODBC 3.8 driver.

For more information, see Compatibility Matrix.
Sample ODBC Program

The ODBC code sample prompts you for an ODBC data source name. You will then be prompted to enter a query and the sample will display the results of the query.

Introduction to ODBC

This section provides a brief history of Structured Query Language and ODBC, and includes conceptual information about the ODBC interface.

This section contains the following topics:

- ODBC Overview
- Introduction to SQL and ODBC
- ODBC Architecture
- ODBC 64-Bit Information

ODBC Overview

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

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

This section contains the following topics.

- Why Was ODBC Created?
- What Is ODBC?
- ODBC and the Standard CLI

Why Was ODBC Created?

Historically, companies used a single DBMS. All database access was done either through the front end of that system or through applications written to work exclusively with that system. However, as the use of computers grew and more computer hardware and software became available, companies started to acquire different DBMSs. The reasons were many: People bought what was cheapest, what was fastest, what they already knew, what was latest on the market, what worked best for a single application. Other reasons were reorganizations and mergers, where departments that previously had a single DBMS now had several.
What Is ODBC?

Many misconceptions about ODBC exist in the computing world. To the end user, it is an icon in the Microsoft® Windows® Control Panel. To the application programmer, it is a library containing data access routines. To many others, it is the answer to all database access problems ever imagined.

First and foremost, ODBC is a specification for a database API. This API is independent of any one DBMS or operating system; although this manual uses C, the ODBC API is language-independent. The ODBC API is based on the CLI specifications from Open Group and ISO/IEC. ODBC 3.x fully implements both of these specifications — earlier versions of ODBC were based on preliminary versions of these specifications but did not fully implement them — and adds features commonly needed by developers of screen-based database applications, such as scrollable cursors.

The functions in the ODBC API are implemented by developers of DBMS-specific drivers. Applications call the functions in these drivers to access data in a DBMS-independent manner. A Driver Manager manages communication between applications and drivers.

Although Microsoft provides a driver manager for computers running Microsoft Windows® 95 and later, has written several ODBC drivers, and calls ODBC functions from some of its applications, anyone can write ODBC applications and drivers. In fact, the vast majority of ODBC applications and drivers available today are written by companies other than Microsoft. Furthermore, ODBC drivers and applications exist on the Macintosh® and a variety of UNIX platforms.

To help application and driver developers, Microsoft offers an ODBC Software Development Kit (SDK) for computers running Windows 95 and later that provides the driver manager, installer DLL, test tools, and sample applications. Microsoft has teamed with Visigenic Software to port these SDKs to the Macintosh and a variety of UNIX platforms.

It is important to understand that ODBC is designed to expose database capabilities, not supplement them. Thus, application writers should not expect that using ODBC will suddenly transform a simple database into a fully featured relational database engine. Nor are driver writers expected to implement functionality not found in the underlying database. An exception to this is that developers who write drivers that directly access file data (such as data in an Xbase file) are required to write a database engine that supports at least minimal SQL functionality. Another exception is that the ODBC component of the Windows SDK, formerly included in the Microsoft Data Access Components (MDAC) SDK, provides a cursor library that simulates scrollable cursors for drivers that implement a certain level of functionality.

Applications that use ODBC are responsible for any cross-database functionality. For example, ODBC is not a heterogeneous join engine, nor is it a distributed transaction processor. However, because it is DBMS-independent, it can be used to build such cross-database tools.

© 2016 Microsoft

ODBC and the Standard CLI

ODBC aligns with the following specifications and standards that deal with the Call-Level Interface (CLI). (The ODBC features are a superset of each of these standards.)

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

As a result of this alignment, the following are true:

- An application written to the Open Group and ISO CLI specifications will work with an ODBC 3.x driver or a standards-compliant driver when it is compiled with the ODBC 3.x header files and linked with ODBC 3.x libraries, and when it gains access to the driver through the ODBC 3.x Driver Manager.
- A driver written to the Open Group and ISO CLI specifications will work with an ODBC 3.x application or a standards-compliant application when it is compiled with the ODBC 3.x header files and linked with ODBC 3.x libraries, and when the application gains access to the driver through the ODBC 3.x Driver Manager. (For more information, see Standards-Compliant Applications and Drivers.)
The Core interface conformance level encompasses all the features in the ISO CLI and all the nonoptional features in the Open Group CLI. Optional features of the Open Group CLI appear in higher interface conformance levels. Because all ODBC 3.x drivers are required to support the features in the Core interface conformance level, the following are true:

- An ODBC 3.x driver will support all the features used by a standards-compliant application.
- An ODBC 3.x application using only the features in ISO CLI and the nonoptional features of the Open Group CLI will work with any standards-compliant driver.

In addition to the call-level interface specifications contained in the ISO/IEC and Open Group CLI standards, ODBC implements the following features. (Some of these features existed in versions of ODBC prior to ODBC 3.x.)

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

© 2016 Microsoft

Introduction to SQL and ODBC

ODBC was created to provide a uniform method of access to different, or heterogeneous, database management systems (DBMSs). This introduction discusses concepts related to the development of ODBC.

This section contains the following topics.

- Structured Query Language (SQL)
- Database Access Architecture

© 2016 Microsoft

Structured Query Language (SQL)

A typical DBMS allows users to store, access, and modify data in an organized, efficient way. Originally, the users of DBMSs were programmers. Accessing the stored data required writing a program in a programming language such as COBOL. While these programs were often written to present a friendly interface to a nontechnical user, access to the data itself required the services of a knowledgeable programmer. Casual access to the data was not practical.

Users were not entirely happy with this situation. While they could access data, it often required convincing a DBMS programmer to write special software. For
example, if a sales department wanted to see the total sales in the previous month by each of its salespeople and wanted this information ranked in order by each salesperson’s length of service in the company, it had two choices: Either a program already existed that allowed the information to be accessed in exactly this way, or the department had to ask a programmer to write such a program. In many cases, this was more work than it was worth, and it was always an expensive solution for one-time, or ad hoc, inquiries. As more and more users wanted easy access, this problem grew larger and larger.

Allowing users to access data on an ad hoc basis required giving them a language in which to express their requests. A single request to a database is defined as a query; such a language is called a query language. Many query languages were developed for this purpose, but one of these became the most popular: Structured Query Language, invented at IBM in the 1970s. It is more commonly known by its acronym, SQL, and is pronounced both as “ess-cue-ell” and as “sequel”; this manual uses the former pronunciation. SQL became an ANSI standard in 1986 and an ISO standard in 1987; it is used today in a great many database management systems.

Although SQL solved the ad hoc needs of users, the need for data access by computer programs did not go away. In fact, most database access still was (and is) programmatic, in the form of regularly scheduled reports and statistical analyses, data entry programs such as those used for order entry, and data manipulation programs, such as those used to reconcile accounts and generate work orders.

These programs also use SQL, using one of the following three techniques:

- **Embedded SQL**, in which SQL statements are embedded in a host language such as C or COBOL.
- **SQL modules**, in which SQL statements are compiled on the DBMS and called from a host language.
- **Call-level interface**, or CLI, which consists of functions called to pass SQL statements to the DBMS and to retrieve results from the DBMS.

**Note**

It is a historical accident that the term call-level interface is used instead of application programming interface (API), another term for the same thing. In the database world, API is used to describe SQL itself: SQL is the API to a DBMS.

Of these choices, embedded SQL is the most commonly used, although most major DBMSs support proprietary CLIs.

This section contains the following topics.

- **Processing an SQL Statement**
- **Embedded SQL**
- **SQL Modules**
- **Call-Level Interfaces**

© 2016 Microsoft

### Processing a SQL Statement

Before discussing the techniques for using SQL programmatically, it is necessary to discuss how an SQL statement is processed. The steps involved are common to all three techniques, although each technique performs them at different times. The following illustration shows the steps involved in processing an SQL statement, which are discussed throughout the rest of this section.

![SQL Statement Processing Diagram](image_url)

To process an SQL statement, a DBMS performs the following five steps:

1. The DBMS first parses the SQL statement. It breaks the statement up into individual words, called tokens, makes sure that the statement has a valid verb and valid clauses, and so on. Syntax errors and misspellings can be detected in this step.
2. The DBMS validates the statement. It checks the statement against the system catalog. Do all the tables named in the statement exist in the database? Do all of the columns exist and are the column names unambiguous? Does the user have the required privileges to execute the statement? Certain semantic errors can be detected in this step.

3. The DBMS generates an access plan for the statement. The access plan is a binary representation of the steps that are required to carry out the statement; it is the DBMS equivalent of executable code.

4. The DBMS optimizes the access plan. It explores various ways to carry out the access plan. Can an index be used to speed a search? Should the DBMS first apply a search condition to Table A and then join it to Table B, or should it begin with the join and use the search condition afterward? Can a sequential search through a table be avoided or reduced to a subset of the table? After exploring the alternatives, the DBMS chooses one of them.

5. The DBMS executes the statement by running the access plan.

The steps used to process an SQL statement vary in the amount of database access they require and the amount of time they take. Parsing an SQL statement does not require access to the database and can be done very quickly. Optimization, on the other hand, is a very CPU-intensive process and requires access to the system catalog. For a complex, multitable query, the optimizer may explore thousands of different ways of carrying out the same query. However, the cost of executing the query inefficiently is usually so high that the time spent in optimization is more than regained in increased query execution speed. This is even more significant if the same optimized access plan can be used over and over to perform repetitive queries.

Embedded SQL

The first technique for sending SQL statements to the DBMS is embedded SQL. Because SQL does not use variables and control-of-flow statements, it is often used as a database sublanguage that can be added to a program written in a conventional programming language, such as C or COBOL. This is a central idea of embedded SQL: placing SQL statements in a program written in a host programming language. Briefly, the following techniques are used to embed SQL statements in a host language:

- Embedded SQL statements are processed by a special SQL precompiler. All SQL statements begin with an introducer and end with a terminator, both of which flag the SQL statement for the precompiler. The introducer and terminator vary with the host language. For example, the introducer is "EXEC SQL" in C and "$SQL( in MUMPS, and the terminator is a semicolon (;) in C and a right parenthesis in MUMPS.

- Variables from the application program, called host variables, can be used in embedded SQL statements wherever constants are allowed. These can be used on input to tailor an SQL statement to a particular situation and on output to receive the results of a query.

- Queries that return a single row of data are handled with a singleton SELECT statement; this statement specifies both the query and the host variables in which to return data.

- Queries that return multiple rows of data are handled with cursors. A cursor keeps track of the current row within a result set. The DECLARE CURSOR statement defines the query, the OPEN statement begins the query processing, the FETCH statement retrieves successive rows of data, and the CLOSE statement ends query processing.

- While a cursor is open, positioned update and positioned delete statements can be used to update or delete the row currently selected by the cursor.

This section contains the following topics.

- Embedded SQL Example
- Compiling an Embedded SQL Program
- Static SQL
- Dynamic SQL

Embedded SQL Example

The following code is a simple embedded SQL program, written in C. The program illustrates many, but not all, of the embedded SQL techniques. The program prompts the user for an order number, retrieves the customer number, salesperson, and status of the order, and displays the retrieved information on the screen.

```c
#include <stdio.h>
#include <sqlca.h>

int main() {
    EXEC SQL INCLUDE SQLCA;
    EXEC SQL BEGIN DECLARE SECTION;
    int OrderID;
    char SalesPerson[10];
    char Status[6];
    /* Employee ID (from user) */
    /* Retrieved customer ID */
    /* Retrieved salesperson name */
    /* Retrieved order status */
    EXEC SQL END DECLARE SECTION;
    /* Set up error processing */
```
Note the following about this program:

- **Host Variables** The host variables are declared in a section enclosed by the `BEGIN DECLARE SECTION` and `END DECLARE SECTION` keywords. Each host variable name is prefixed by a colon (:) when it appears in an embedded SQL statement. The colon allows the precompiler to distinguish between host variables and database objects, such as tables and columns, that have the same name.

- **Data Types** The data types supported by a DBMS and a host language can be quite different. This affects host variables because they play a dual role. On one hand, host variables are program variables, declared and manipulated by host language statements. On the other hand, they are used in embedded SQL statements to retrieve database data. If there is no host language type that corresponds to a DBMS data type, the DBMS automatically converts the data. However, because each DBMS has its own rules and idiosyncrasies associated with the conversion process, the host variable types must be chosen carefully.

- **Error Handling** The DBMS reports run-time errors to the applications program through an SQL Communications Area, or SQLCA. In the preceding code example, the first embedded SQL statement is `INCLUDE SQLCA`. This tells the precompiler to include the SQLCA structure in the program. This is required whenever the program will process errors returned by the DBMS. The `WHENEVER...GOTO` statement tells the precompiler to generate error-handling code that branches to a specific label when an error occurs.

- **Singleton SELECT** The statement used to return the data is a singleton SELECT statement; that is, it returns only a single row of data. Therefore, the code example does not declare or use cursors.

© 2016 Microsoft

### Compiling an Embedded SQL Program

Because an embedded SQL program contains a mix of SQL and host language statements, it cannot be submitted directly to a compiler for the host language. Instead, it is compiled through a multistep process. Although this process differs from product to product, the steps are roughly the same for all products.

This illustration shows the steps necessary to compile an embedded SQL program.
Five steps are involved in compiling an embedded SQL program:

1. The embedded SQL program is submitted to the SQL precompiler, a programming tool. The precompiler scans the program, finds the embedded SQL statements, and processes them. A different precompiler is required for each programming language supported by the DBMS. DBMS products typically offer precompilers for one or more languages, including C, Pascal, COBOL, Fortran, Ada, PL/I, and various assembly languages.

2. The precompiler produces two output files. The first file is the source file, stripped of its embedded SQL statements. In their place, the precompiler substitutes calls to proprietary DBMS routines that provide the run-time link between the program and the DBMS. Typically, the names and the calling sequences of these routines are known only to the precompiler and the DBMS; they are not a public interface to the DBMS. The second file is a copy of all the embedded SQL statements used in the program. This file is sometimes called a database request module, or DBRM.

3. The source file output from the precompiler is submitted to the standard compiler for the host programming language (such as a C or COBOL compiler). The compiler processes the source code and produces object code as its output. Note that this step has nothing to do with the DBMS or with SQL.

4. The linker accepts the object modules generated by the compiler, links them with various library routines, and produces an executable program. The library routines linked into the executable program include the proprietary DBMS routines described in step 2.

5. The database request module generated by the precompiler is submitted to a special binding utility. This utility examines the SQL statements, parses, validates, and optimizes them, and then produces an access plan for each statement. The result is a combined access plan for the entire program, representing an executable version of the embedded SQL statements. The binding utility stores the plan in the database, usually assigning it the name of the application program that will use it. Whether this step takes place at compile time or run time depends on the DBMS.

Notice that the steps used to compile an embedded SQL program correlate very closely with the steps described earlier in Processing an SQL Statement. In particular, notice that the precompiler separates the SQL statements from the host language code, and the binding utility parses and validates the SQL statements and creates the access plans. In DBMSs where step 5 takes place at compile time, the first four steps of processing an SQL statement take place at compile time, while the last step (execution) takes place at run time. This has the effect of making query execution in such DBMSs very fast.

© 2016 Microsoft

Static SQL

The embedded SQL shown in Embedded SQL Example is known as static SQL. It is called static SQL because the SQL statements in the program are static; that is, they do not change each time the program is run. As described in the previous section, these statements are compiled when the rest of the program is compiled.

Static SQL works well in many situations and can be used in any application for which the data access can be determined at program design time. For example, an order-entry program always uses the same statement to insert a new order, and an airline reservation system always uses the same statement to change the status of a seat from available to reserved. Each of these statements would be generalized through the use of host variables; different values can be inserted in a sales order, and different seats can be reserved. Because such statements can be hard-coded in the program, such programs have the advantage that the statements need to be parsed, validated, and optimized only once, at compile time. This results in relatively fast code.

© 2016 Microsoft

Dynamic SQL

Although static SQL works well in many situations, there is a class of applications in which the data access cannot be determined in advance. For example,
suppose a spreadsheet allows a user to enter a query, which the spreadsheet then sends to the DBMS to retrieve data. The contents of this query obviously cannot be known to the programmer when the spreadsheet program is written.

To solve this problem, the spreadsheet uses a form of embedded SQL called dynamic SQL. Unlike static SQL statements, which are hard-coded in the program, dynamic SQL statements can be built at run time and placed in a string host variable. They are then sent to the DBMS for processing. Because the DBMS must generate an access plan at run time for dynamic SQL statements, dynamic SQL is generally slower than static SQL. When a program containing dynamic SQL statements is compiled, the dynamic SQL statements are not stripped from the program, as in static SQL. Instead, they are replaced by a function call that passes the statement to the DBMS; static SQL statements in the same program are treated normally.

The simplest way to execute a dynamic SQL statement is with an EXECUTE IMMEDIATE statement. This statement passes the SQL statement to the DBMS for compilation and execution.

One disadvantage of the EXECUTE IMMEDIATE statement is that the DBMS must go through each of the five steps of processing an SQL statement each time the statement is executed. The overhead involved in this process can be significant if many statements are executed dynamically, and it is wasteful if those statements are similar. To address this situation, dynamic SQL offers an optimized form of execution called prepared execution, which uses the following steps:

1. The program constructs an SQL statement in a buffer, just as it does for the EXECUTE IMMEDIATE statement. Instead of host variables, a question mark (?) can be substituted for a constant anywhere in the statement text to indicate that a value for the constant will be supplied later. The question mark is called as a parameter marker.

2. The program passes the SQL statement to the DBMS with a PREPARE statement, which requests that the DBMS parse, validate, and optimize the statement and generate an execution plan for it. The program then uses an EXECUTE statement (not an EXECUTE IMMEDIATE statement) to execute the PREPARE statement at a later time. It passes parameter values for the statement through a special data structure called the SQL Data Area or SQLDA.

3. The program can use the EXECUTE statement repeatedly, supplying different parameter values each time the dynamic statement is executed.

Prepared execution is still not the same as static SQL. In static SQL, the first four steps of processing an SQL statement take place at compile time. In prepared execution, these steps still take place at run time, but they are performed only once; execution of the plan takes place only when EXECUTE is called. This helps eliminate some of the performance disadvantages inherent in the architecture of dynamic SQL. The next illustration shows the differences between static SQL, dynamic SQL with immediate execution, and dynamic SQL with prepared execution.

### SQL Modules

The second technique for sending SQL statements to the DBMS is through modules. Briefly, a module consists of a group of procedures, which are called from the host programming language. Each procedure contains a single SQL statement, and data is passed to and from the procedure through parameters.

A module can be thought of as an object library that is linked to the application code. However, exactly how the procedures and the rest of the application are linked is implementation-dependent. For example, the procedures could be compiled into object code and linked directly to the application code, they could be compiled and stored on the DBMS and calls to access plan identifiers placed in the application code, or they could be interpreted at run time.

The main advantage of modules is that they cleanly separate SQL statements from the programming language. In theory, it should be possible to change one without changing the other and simply relink them.

### Call–Level Interfaces

The final technique for sending SQL statements to the DBMS is through a call–level interface (CLI). A call–level interface provides a library of DBMS functions that can be called by the application program. Thus, instead of trying to blend SQL with another programming language, a call–level interface is similar to the routine libraries most programmers are accustomed to using, such as the string, I/O, or math libraries in C. Note that DBMSs that support embedded SQL already have a call–level interface, the calls to which are generated by the precompiler. However, these calls are undocumented and subject to change without notice.

Call–level interfaces are commonly used in client/server architectures, in which the application program (the client) resides on one computer and the DBMS (the server) resides on a different computer. The application calls CLI functions on the local system, and those calls are sent across the network to the DBMS for processing.

A call–level interface is similar to dynamic SQL, in that SQL statements are passed to the DBMS for processing at run time, but it differs from embedded SQL as a whole in that there are no embedded SQL statements and no precompiler is required.

Using a call–level interface typically involves the following steps:

1. The application calls a CLI function to connect to the DBMS.
2. The application builds an SQL statement and places it in a buffer. It then calls one or more CLI functions to send the statement to the DBMS for preparation and execution.
3. If the statement is a SELECT statement, the application calls a CLI function to return the results in application buffers. Typically, this function returns one row or one column of data at a time.
4. The application calls a CLI function to disconnect from the DBMS.
Database Access Architecture

One of the questions in the development of ODBC was which part of the database access architecture to standardize. The SQL programming interfaces described in the previous section — embedded SQL, SQL modules, and CLIs — are only one part of this architecture. In fact, because ODBC was primarily intended to connect personal computer–based applications to minicomputer and mainframe DBMSs, there were also a number of network components, some of which could be standardized.

This section contains the following topics.

- Network Database Access
- Standard Database Access Architectures
- The ODBC Solution

Network Database Access

Accessing a database across a network requires a number of components, each of which is independent of, and resides beneath, the programming interface. These components are shown in the following illustration.

- **Programming Interface** As described earlier in this section, the programming interface contains the calls made by the application. These interfaces (embedded SQL, SQL modules, and call-level interfaces) are generally specific to each DBMS, although they are usually based on an ANSI or ISO standard.

- **Data Stream Protocol** The data stream protocol describes the stream of data transferred between the DBMS and its client. For example, the protocol might require the first byte to describe what the rest of the stream contains: an SQL statement to be executed, a returned error value, or returned data. The format of the rest of the data in the stream would then depend on this flag. For example, an error stream might contain the flag, a 2-byte integer error code, a 2-byte integer error message length, and an error message.

  The data stream protocol is a logical protocol and is independent of the protocols used by the underlying network. Thus, a single data stream protocol can generally be used on a number of different networks. Data stream protocols are typically proprietary and have been optimized to work with a particular DBMS.

- **Interprocess Communication Mechanism** The interprocess communication (IPC) mechanism is the process by which one process communicates with another. Examples include named pipes, TCP/IP sockets, and DECnet sockets. The choice of IPC mechanism is constrained by the operating system and network being used.

- **Network Protocol** The network protocol is used to transport the data stream over a network. It can be considered the plumbing that supports the IPC mechanisms used to implement the data stream protocol, as well as supporting basic network operations such as file transfers and print sharing. Network protocols include NetBEUI, TCP/IP, DECnet, and SPX/IPX and are specific to each network.

Standard Database Access Architectures
In looking at the database access components described in the preceding section, it turns out that two of them — programming interfaces and data stream protocols — are good candidates for standardization. The other two components — IPC mechanism and network protocols — not only reside at too low a level but they are both highly dependent on the network and operating system. There is also a third approach — gateways — that provides possibilities for standardization.

This section contains the following topics.

- Standard Programming Interface
- Standard Data Stream Protocol
- Standard Gateway

Standard Programming Interface

The programming interface is perhaps the most obvious candidate for standardization. In fact, when ODBC was being developed, ANSI and ISO already provided standards for embedded SQL and SQL modules. Although no standards existed for a database CLI, the SQL Access Group — an industry consortium of database vendors — was considering whether to create one; parts of ODBC later became the basis for their work.

One of the requirements for ODBC was that a single application binary had to work with multiple DBMSs. It is for this reason that ODBC does not use embedded SQL or module languages. Although the language in embedded SQL and module languages is standardized, each is tied to DBMS-specific precompilers. Thus, applications must be recompiled for each DBMS and the resulting binaries work only with a single DBMS. While this is acceptable for the low-volume applications found in the minicomputer and mainframe worlds, it is unacceptable in the personal computer world. First, it is a logistical nightmare to deliver multiple versions of high-volume, shrink-wrapped software to customers; second, personal computer applications often need to access multiple DBMS simultaneously.

On the other hand, a call-level interface can be implemented through libraries, or database drivers, that reside on each local machine; a different driver is required for each DBMS. Because modern operating systems can load such libraries (such as dynamic-link libraries on the Microsoft® Windows® operating system) at run time, a single application can access data from different DBMSs without recompilation and can also access data from multiple databases simultaneously. As new database drivers become available, users can just install these on their computers without having to modify, recompile, or relink their database applications. Furthermore, a call-level interface was a good candidate for ODBC because Windows — the platform for which ODBC was originally developed — already made extensive use of such libraries.

Standard Data Stream Protocol


Standard Gateway

A gateway is a piece of software that causes one DBMS to look like another. That is, the gateway accepts the programming interface, SQL grammar, and data stream protocol of a single DBMS and translates it to the programming interface, SQL grammar, and data stream protocol of the hidden DBMS. For example, applications written to use Microsoft® SQL Server™ can also access DB2 data through the Micro Decisionware DB2 Gateway; this product causes DB2 to look like SQL Server. When gateways are used, a different gateway must be written for each target database.

Although gateways are limited by architectural differences among DBMSs, they are a good candidate for standardization. However, if all DBMSs are to standardize on the programming interface, SQL grammar, and data stream protocol of a single DBMS, whose DBMS is to be chosen as the standard? Certainly no commercial DBMS vendor is likely to agree to standardize on a competitor's product. And if a standard programming interface, SQL grammar, and data stream protocol are developed, no gateway is needed.

The ODBC Solution
The question, then, is how does ODBC standardize database access? There are two architectural requirements:

- Applications must be able to access multiple DBMSs using the same source code without recompiling or relinking.
- Applications must be able to access multiple DBMSs simultaneously.

And there is one more question, due to marketplace reality:

- Which DBMS features should ODBC expose? Only features that are common to all DBMSs or any feature that is available in any DBMS?

ODBC solves these problems in the following manner:

- **ODBC is a call-level interface.** To solve the problem of how applications access multiple DBMSs using the same source code, ODBC defines a standard CLI. This contains all of the functions in the CLI specifications from Open Group and ISO/IEC and provides additional functions commonly required by applications. A different library, or driver, is required for each DBMS that supports ODBC. The driver implements the functions in the ODBC API. To use a different driver, the application does not need to be recompiled or relinked. Instead, the application simply loads the new driver and calls the functions in it. To access multiple DBMSs simultaneously, the application loads multiple drivers. How drivers are supported is operating system–specific. For example, on the Microsoft® Windows® operating system, drivers are dynamic-link libraries (DLLs).

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

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

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

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

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

- **ODBC exposes a significant number of DBMS features but does not require drivers to support all of them.** If ODBC exposed only features that are common to all DBMSs, it would be of little use; after all, the reason so many different DBMSs exist today is that they have different features. If ODBC exposed every feature that is available in any DBMS, it would be impossible for drivers to implement.

Instead, ODBC exposes a significant number of features — more than are supported by most DBMSs — but requires drivers to implement only a subset of those features. Drivers implement the remaining features only if they are supported by the underlying DBMS or if they choose to emulate them. Thus, applications can be written to exploit the features of a single DBMS as exposed by the driver for that DBMS, to use only those features used by all DBMSs, or to check for support of a particular feature and react accordingly.

So that an application can determine what features a driver and DBMS support, ODBC provides two functions (SQLGetInfo and SQLGetFunctions) that return general information about the driver and DBMS capabilities and a list of functions the driver supports. ODBC also defines API and SQL grammar conformance levels, which specify broad ranges of features supported by the driver. For more information, see Conformance Levels.

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

**ODBC Architecture**

The ODBC architecture has four components:

- **Application** Performs processing and calls ODBC functions to submit SQL statements and retrieve results.
- **Driver Manager** Loads and unloads drivers on behalf of an application. Processes ODBC function calls or passes them to a driver.
- **Driver** Processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application. If necessary, the driver modifies an application's request so that the request conforms to syntax supported by the associated DBMS.
• **Data Source** Consists of the data the user wants to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS.

Note the following points about the ODBC architecture. First, multiple drivers and data sources can exist, which allows the application to simultaneously access data from more than one data source. Second, the ODBC API is used in two places: between the application and the Driver Manager, and between the Driver Manager and each driver. The interface between the Driver Manager and the drivers is sometimes referred to as the service provider interface, or SPI. For ODBC, the application programming interface (API) and the service provider interface (SPI) are the same; that is, the Driver Manager and each driver have the same interface to the same functions.

This section contains the following topics.

- **Applications**
- **The Driver Manager**
- **Drivers**
- **Data Sources**

© 2016 Microsoft

### Applications

An application is a program that calls the ODBC API to access data. Although many types of applications are possible, most fall into three categories, which are used as examples throughout this guide.

- **Generic Applications** These are also referred to as shrink-wrapped applications or off-the-shelf applications. Generic applications are designed to work with a variety of different DBMSs. Examples include a spreadsheet or statistics package that uses ODBC to import data for further analysis and a word processor that uses ODBC to get a mailing list from a database.

  An important subcategory of generic applications is application development environments, such as PowerBuilder or Microsoft® Visual Basic®. Although the applications constructed with these environments will probably work only with a single DBMS, the environment itself needs to work with multiple DBMSs.

  What all generic applications have in common is that they are highly interoperable among DBMSs and they need to use ODBC in a relatively generic manner. For more information about interoperability, see Choosing a Level of Interoperability.

- **Vertical Applications** Vertical applications perform a single type of task, such as order entry or tracking manufacturing data, and work with a database schema that is controlled by the developer of the application. For a particular customer, the application works with a single DBMS. For example, a small business might use the application with dBase, while a large business might use it with Oracle.

  The application uses ODBC in such a manner that the application is not tied to any one DBMS, although it might be tied to a limited number of DBMSs that provide similar functionality. Thus, the application developer can sell the application independently from the DBMS. Vertical applications are interoperable when they are developed but are sometimes modified to include noninteroperable code once the customer has chosen a DBMS.

- **Custom Applications** Custom applications are used to perform a specific task in a single company. For example, an application in a large company might gather sales data from several divisions (each of which uses a different DBMS) and create a single report. ODBC is used because it is a common interface and saves programmers from having to learn multiple interfaces. Such applications are generally not interoperable and are written to specific DBMSs and drivers.

A number of tasks are common to all applications, no matter how they use ODBC. Taken together, they largely define the flow of any ODBC application. The tasks are:

- Selecting a data source and connecting to it.
- Submitting an SQL statement for execution.
- Retrieving results (if any).
- Processing errors.
- Committing or rolling back the transaction enclosing the SQL statement.
- Disconnecting from the data source.

Because most data access work is done with SQL, the primary task for which applications use ODBC is to submit SQL statements and retrieve the results (if any) generated by those statements. Other tasks for which applications use ODBC include determining and adjusting to driver capabilities and browsing the database catalog.

© 2016 Microsoft

### The Driver Manager
The Driver Manager is a library that manages communication between applications and drivers. For example, on Microsoft® Windows® platforms, the Driver Manager is a dynamic-link library (DLL) that is written by Microsoft and can be redistributed by users of the redistributable MDAC 2.8 SP1 SDK.

The Driver Manager exists mainly as a convenience to application writers and solves a number of problems common to all applications. These include determining which driver to load based on a data source name, loading and unloading drivers, and calling functions in drivers.

To see why the latter is a problem, consider what would happen if the application called functions in the driver directly. Unless the application was linked directly to a particular driver, it would have to build a table of pointers to the functions in that driver and call those functions by pointer. Using the same code for more than one driver at a time would add yet another level of complexity. The application would first have to set a function pointer to point to the correct function in the correct driver, and then call the function through that pointer.

The Driver Manager solves this problem by providing a single place to call each function. The application is linked to the Driver Manager and calls ODBC functions in the Driver Manager, not the driver. The application identifies the target driver and data source with a connection handle. When it loads a driver, the Driver Manager builds a table of pointers to the functions in that driver. It uses the connection handle passed by the application to find the address of the function in the target driver and calls that function by address.

For the most part, the Driver Manager just passes function calls from the application to the correct driver. However, it also implements some functions (SQLDataSources, SQLDrivers, and SQLGetFunctions) and performs basic error checking. For example, the Driver Manager checks that handles are not null pointers, that functions are called in the correct order, and that certain function arguments are valid. For a complete description of the errors checked by the Driver Manager, see the reference section for each function and Appendix B: ODBC State Transition Tables.

The final major role of the Driver Manager is loading and unloading drivers. The application loads and unloads only the Driver Manager. When it wants to use a particular driver, it calls a connection function (SQLConnect, SQLDriverConnect, or SQLBrowseConnect) in the Driver Manager and specifies the name of a particular data source or driver, such as "Accounting" or "SQL Server." Using this name, the Driver Manager searches the data source information for the driver's file name, such as Sqlsrvr.dll. It then loads the driver (assuming it is not already loaded), stores the address of each function in the driver, and calls the connection function in the driver, which then initializes itself and connects to the data source.

When the application is done using the driver, it calls SQLDisconnect in the Driver Manager. The Driver Manager calls this function in the driver, which disconnects from the data source. However, the Driver Manager keeps the driver in memory in case the application reconnections to it. It unloads the driver only when the application frees the connection used by the driver or uses the connection for a different driver, and no other connections use the driver. For a complete description of the Driver Manager's role in loading and unloading drivers, see Driver Manager's Role in the Connection Process.

Drivers

Drivers are libraries that implement the functions in the ODBC API. Each is specific to a particular DBMS; for example, a driver for Oracle cannot directly access data in an Informix DBMS. Drivers expose the capabilities of the underlying DBMSs; they are not required to implement capabilities not supported by the DBMS. For example, if the underlying DBMS does not support outer joins, then neither should the driver. The only major exception to this is that drivers for DBMSs that do not have stand-alone database engines, such as Xbase, must implement a database engine that at least supports a minimal amount of SQL.

This section contains the following topics.

- Driver Tasks
- Driver Architecture

See Also

Microsoft-Supplied ODBC Drivers

Driver Tasks

Specific tasks performed by drivers include:

- Connecting to and disconnecting from the data source.
- Checking for function errors not checked by the Driver Manager.
- Initiating transactions; this is transparent to the application.
- Submitting SQL statements to the data source for execution. The driver must modify ODBC SQL to DBMS-specific SQL; this is often limited to replacing escape clauses defined by ODBC with DBMS-specific SQL.
- Sending data to and retrieving data from the data source, including converting data types as specified by the application.
- Mapping DBMS-specific errors to ODBC SQLSTATEs.
Driver Architecture

Driver architecture falls into two categories, depending on which software processes SQL statements:

- **File-Based Drivers** The driver accesses the physical data directly. In this case, the driver acts as both driver and data source; that is, it processes ODBC calls and SQL statements. For example, dBASE drivers are file-based drivers because dBASE does not provide a stand-alone database engine the driver can use. It is important to note that developers of file-based drivers must write their own database engines.

- **DBMS-Based Drivers** The driver accesses the physical data through a separate database engine. In this case the driver processes only ODBC calls; it passes SQL statements to the database engine for processing. For example, Oracle drivers are DBMS-based drivers because Oracle has a stand-alone database engine the driver uses. Where the database engine resides is immaterial. It can reside on the same machine as the driver or a different machine on the network; it might even be accessed through a gateway.

Driver architecture is generally interesting only to driver writers; that is, driver architecture generally makes no difference to the application. However, the architecture can affect whether an application can use DBMS-specific SQL. For example, Microsoft Access provides a stand-alone database engine. If a Microsoft Access driver is DBMS-based — it accesses the data through this engine — the application can pass Microsoft Access–SQL statements to the engine for processing.

However, if the driver is file-based — that is, it contains a proprietary engine that accesses the Microsoft® Access .mdb file directly — any attempts to pass Microsoft Access–specific SQL statements to the engine are likely to result in syntax errors. The reason is that the proprietary engine is likely to implement only ODBC SQL.

This section contains the following topics.

- File-Based Drivers
- DBMS-Based Drivers
- Network Example
- Other Driver Architectures

File-Based Drivers

File-based drivers are used with data sources such as dBASE that do not provide a stand-alone database engine for the driver to use. These drivers access the physical data directly and must implement a database engine to process SQL statements. As a standard practice, the database engines in file-based drivers implement the subset of ODBC SQL defined by the minimum SQL conformance level; for a list of the SQL statements in this conformance level, see Appendix C: SQL Grammar.

In comparing file-based and DBMS-based drivers, file-based drivers are harder to write because of the database engine component, less complicated to configure because there are no network pieces, and less powerful because few people have the time to write database engines as powerful as those produced by database companies.

The following illustration shows two different configurations of file-based drivers, one in which the data resides locally and the other in which it resides on a network file server.

DBMS-Based Drivers

DBMS-based drivers are used with data sources such as Oracle or SQL Server that provide a stand-alone database engine for the driver to use. These drivers access the physical data through the stand-alone engine; that is, they submit SQL statements to and retrieve results from the engine.
Because DBMS-based drivers use an existing database engine, they are usually easier to write than file-based drivers. Although a DBMS-based driver can be easily implemented by translating ODBC calls to native API calls, this results in a slower driver. A better way to implement a DBMS-based driver is to use the underlying data stream protocol, which is usually what the native API does. For example, a SQL Server driver should use TDS (the data stream protocol for SQL Server) rather than DB Library (the native API for SQL Server). An exception to this rule is when ODBC is the native API. For example, Watcom SQL is a stand-alone engine that resides on the same machine as the application and is loaded directly as the driver.

DBMS-based drivers act as the client in a client/server configuration where the data source acts as the server. In most cases, the client (driver) and server (data source) reside on different machines, although both could reside on the same machine running a multitasking operating system. A third possibility is a gateway, which sits between the driver and data source. A gateway is a piece of software that causes one DBMS to look like another. For example, applications written to use SQL Server can also access DB2 data through the Micro Decisionware DB2 Gateway; this product causes DB2 to look like SQL Server.

The following illustration shows three different configurations of DBMS-based drivers. In the first configuration, the driver and data source reside on the same machine. In the second, the driver and data source reside on different machines. In the third, the driver and data source reside on different machines and a gateway sits between them, residing on yet another machine.

**Network Example**

This illustration shows how each of the preceding configurations could appear in a single network.

**Other Driver Architectures**
Some ODBC drivers do not strictly conform to the architecture described previously. This might be because the drivers perform duties other than those of a traditional ODBC driver, or are not drivers in the normal sense.

**Driver as a Middle Component**

The ODBC driver may reside between the Driver Manager and one or more other ODBC drivers. When the driver in the middle is capable of working with multiple data sources, it acts as a dispatcher of ODBC calls (or appropriately translated calls) to other modules that actually access the data sources. In this architecture, the driver in the middle is taking on some of the role of a Driver Manager.

Another example of this sort of driver is a spy program for ODBC, which intercepts and copies ODBC functions being sent between the Driver Manager and the driver. This layer can be used to emulate either a driver or an application. To the Driver Manager, the layer appears to be the driver; to the driver, the layer appears to be the Driver Manager.

**Heterogeneous Join Engines**

Some ODBC drivers are built upon a query engine for performing heterogeneous joins. In one architecture of a heterogeneous join engine (see the following illustration), the driver appears to the application as a driver but appears to another instance of the Driver Manager as an application. This driver processes a heterogeneous join from the application by calling separate SQL statements in drivers for each joined database.

This architecture provides a common interface for the application to access data from different databases. It can use a common way to retrieve metadata, such as information about special columns (row identifiers), and it can call common catalog functions to retrieve data dictionary information. By calling the ODBC function `SQLStatistics`, for instance, the application can retrieve information about the indexes on the tables to be joined, even if the tables are on two separate databases. The query processor does not have to worry about how the databases store metadata.

The application also has standard access to data types. ODBC defines common SQL data types that DBMS-specific data types are mapped to. An application can call `SQLGetTypeInfo` to retrieve information about data types on different databases.

When the application generates a heterogeneous join statement, the query processor in this architecture parses the SQL statement and then generates separate SQL statements for each database to be joined. By using metadata about each driver, the query processor can determine the most efficient, intelligent join. For example, if the statement joins two tables on one database with one table on another database, the query processor can join the two tables on the one database before joining the result with the table from the other database.

**ODBC on the Server**

ODBC drivers can be installed on a server so that they can be used by applications on any of a series of client machines. In this architecture (see the following illustration), a Driver Manager and a single ODBC driver are installed on each client, and another Driver Manager and a series of ODBC drivers are installed on the server. This allows each client access to a variety of drivers used and maintained on the server.
One advantage of this architecture is efficient software maintenance and configuration. Drivers need only be updated in one place: on the server. By using system data sources, data sources can be defined on the server for use by all clients. The data sources need not be defined on the client. Connection pooling can be used to streamline the process by which clients connect to data sources.

The driver on the client is usually a very small driver that transfers the Driver Manager call to the server. Its footprint can be significantly smaller than the fully functional ODBC drivers on the server. In this architecture, client resources can be freed if the server has more computing power. In addition, the efficiency and security of the entire system can be enhanced by installing backup servers and performing load balancing to optimize server use.

© 2016 Microsoft

Data Sources

A data source is simply the source of the data. It can be a file, a particular database on a DBMS, or even a live data feed. The data might be located on the same computer as the program, or on another computer somewhere on a network. For example, a data source might be an Oracle DBMS running on an OS/2® operating system, accessed by Novell® Netware; an IBM DB2 DBMS accessed through a gateway; a collection of Xbase files in a server directory; or a local Microsoft® Access database file.

The purpose of a data source is to gather all of the technical information needed to access the data — the driver name, network address, network software, and so on — into a single place and hide it from the user. The user should be able to look at a list that includes Payroll, Inventory, and Personnel, choose Payroll from the list, and have the application connect to the payroll data, all without knowing where the payroll data resides or how the application got to it.

The term data source should not be confused with similar terms. In this manual, DBMS or database refers to a database program or engine. A further distinction is made between desktop databases, designed to run on personal computers and often lacking in full SQL and transaction support, and server databases, designed to run in a client/server situation and characterized by a stand-alone database engine and rich SQL and transaction support. Database also refers to a particular collection of data, such as a collection of Xbase files in a directory or a database on SQL Server. It is generally equivalent to the term catalog, used elsewhere in this manual, or the term qualifier in earlier versions of ODBC.

This section contains the following topics.

- Types of Data Sources
- Using Data Sources
- Data Source Example

© 2016 Microsoft

Types of Data Sources

There are two types of data sources: machine data sources and file data sources. Although both contain similar information about the source of the data, they differ in the way this information is stored. Because of these differences, they are used in somewhat different manners.

This section contains the following topics.

- Machine Data Sources
- File Data Sources

© 2016 Microsoft
Machine data sources are stored on the system with a user-defined name. Associated with the data source name is all of the information the Driver Manager and driver need to connect to the data source. For an Xbase data source, this might be the name of the Xbase driver, the full path of the directory containing the Xbase files, and some options that tell the driver how to use those files, such as single-user mode or read-only. For an Oracle data source, this might be the name of the Oracle driver, the server where the Oracle DBMS resides, the SQL*Net connection string that identifies the SQL*Net driver to use, and the system ID of the database on the server.

File Data Sources

File data sources are stored in a file and allow connection information to be used repeatedly by a single user or shared among several users. When a file data source is used, the Driver Manager makes the connection to the data source using the information in a .dsn file. This file can be manipulated like any other file. A file data source does not have a data source name, as does a machine data source, and is not registered to any one user or machine.

A file data source streamlines the connection process, because the .dsn file contains the connection string that would otherwise have to be built for a call to the SQLDriverConnect function. Another advantage of the .dsn file is that it can be copied to any machine, so identical data sources can be used by many machines as long as they have the appropriate driver installed. A file data source can also be shared by applications. A shareable file data source can be placed on a network and used simultaneously by multiple applications.

A .dsn file can also be unshareable. An unshareable .dsn file resides on a single machine and points to a machine data source. Unshareable file data sources exist mainly to allow the easy conversion of machine data sources to file data sources so that an application can be designed to work solely with file data sources. When the Driver Manager is sent the information in an unshareable file data source, it connects as necessary to the machine data source that the .dsn file points to.

For more information about file data sources, see Connecting Using File Data Sources, or the SQLDriverConnect function description.

Using Data Sources

Data sources usually are created by the end user or a technician with a program called the ODBC Administrator. The ODBC Administrator prompts the user for the driver to use and then calls that driver. The driver displays a dialog box that requests the information it needs to connect to the data source. After the user enters the information, the driver stores it on the system.

Later, the application calls the Driver Manager and passes it the name of a machine data source or the path of a file containing a file data source. When passed a machine data source name, the Driver Manager searches the system to find the driver used by the data source. It then loads the driver and passes the data source name to it. The driver uses the data source name to find the information it needs to connect to the data source. Finally, it connects to the data source, typically prompting the user for a user ID and password, which generally are not stored.

When passed a file data source, the Driver Manager opens the file and loads the specified driver. If the file also contains a connection string, it passes this to the driver. Using the information in the connection string, the driver connects to the data source. If no connection string was passed, the driver generally prompts the user for the necessary information.

Data Source Example

On computers running Microsoft® Windows NT® Server/Windows 2000 Server, Microsoft Windows NT Workstation/Windows 2000 Professional, or Microsoft Windows® 95/98, machine data source information is stored in the registry. Depending on which registry key the information is stored under, the data source is known as a user data source or a system data source. User data sources are stored under the HKEY_CURRENT_USER key and are available only to the current user. System data sources are stored under the HKEY_LOCAL_MACHINE key and can be used by more than one user on one machine. They can also be used by systemwide services, which can then gain access to the data source even if no user is logged on to the machine. For more information about user and system data sources, see SQLManageDataSources.

Suppose a user has three user data sources: Personnel and Inventory, which use an Oracle DBMS; and Payroll, which uses a Microsoft SQL Server DBMS. The registry values for data sources might be:

| HKEY_CURRENT_USER | SOFTWARE | ODBC | Odbc.ini | ODBC Data Sources | Personnel: REG_SZ: Oracle | Inventory: REG_SZ: Oracle |
and the registry values for the Payroll data source might be:

```
HKEY_CURRENT_USER
  SOFTWARE
    ODBC
      Odbc.ini
        Payroll
          Driver : REG_SZ : C:\WINDOWS\SYSTEM\Sqlsrvr.dll
          Description : REG_SZ : Payroll database
          Server : REG_SZ : PYRLL1
          UseProcForPrepare : REG_SZ : Yes
          FastConnectOption : REG_SZ : No
          OEMTOANSI : REG_SZ : No
          LastUser : REG_SZ : smithjo
          Database : REG_SZ : Payroll
          Language : REG_SZ :
```

© 2016 Microsoft

ODBC 64–Bit Information

Beginning with Windows Server 2003, Microsoft operating systems have supported the 64-bit ODBC libraries. The ODBC headers and libraries first shipped with MDAC 2.7 SDK contain changes to allow programmers to easily write code for the new 64-bit platforms. By ensuring that your code uses the ODBC defined types listed below, you can compile the same source code both for 64-bit and 32-bit platforms based on the _WIN64 or _WIN32 macros.

There are several points to keep in mind when programming for a 64-bit processor:

- Although the size of a pointer has changed from 4 bytes to 8 bytes, integers and longs are still 4 byte values. The types INT64 and UINT64 have been defined for 8 byte integers. The new ODBC types SQLLEN and SQLULEN are defined in the ODBC header file as INT64 and UINT64 when _WIN64 has been defined.

- Several functions in ODBC are declared as taking a pointer parameter. In 32-bit ODBC, parameters defined as pointers were frequently used to pass either an integer value or a pointer to a buffer depending on the context of the call. This was, of course, possible due to the fact that pointers and integers had the same size. In 64-bit Windows, this is not the case.

- Some ODBC functions that were previously defined with SQLINTEGER and SQLUINTEGER parameters have been changed where appropriate to use the new SQLLEN and SQLULEN typedefs. These changes are listed in the next section, Function Declaration Changes.

- Some of the descriptor fields that can be set through the various SQLSet and SQLGet functions have been changed to accommodate 64-bit values while others are still 32-bit values. Make sure that you use the appropriate sized variable when setting and retrieving these fields. Specifics of which descriptor fields have changed are listed under Function Declaration Changes.

Function Declaration Changes

The following function signatures have changed for 64-bit programming. The items in bold text are the specific parameters that are different.

```
SQLBindCol (SQLHSTMT StatementHandle, SQLUSMALLINT ColumnNumber,
            SQLSMALLINT ValueType, SQLSMALLINT ParameterType,
            SQLULEN ColumnSize, SQLSMALLINT DecimalDigits,
            SQLPOINTER ParameterValuePtr, SQLLEN *Strien_or_Ind);

SQLBindParam (SQLHSTMT StatementHandle, SQuSMALLINT ParameterNumber,
              SQLSMALLINT ValueType, SQLSMALLINT ParameterType,
              SQLULEN ColumnSize, SQLSMALLINT DecimalDigits,
              SQLPOINTER ParameterValuePtr, SQLLEN *Strien_or_Ind);

SQLBindParameter (SQLHSTMT StatementHandle, SQLSMALLINT ParameterNumber,
                  SQLSMALLINT InputOutputType, SQLSMALLINT ValueType,
                  SQLSMALLINT ParameterType, SQLULEN ColumnSize, SQLSMALLINT DecimalDigits,
                  SQLPOINTER ParameterValuePtr, SQLLEN BufferLength, SQLLEN *Strien_or_IndPtr);

SQLColAttribute (SQLHSTMT StatementHandle, SQLSMALLINT ColumnNumber,
                 SQLSMALLINT FieldIdentifier, SQLPOINTER CharacterAttributePtr,
                 SQLSMALLINT BufferLength, SQLSMALLINT *StringLengthPtr,
                 SQLLEN *NumericAttributePtr);

SQLColAttributes (SQLHSTMT Hstmt, SQLSMALLINT icol,
                  SQLSMALLINT FDescType, SQLPOINTER rgbDesc,
                  SQLSMALLINT cDescMax, SQLSMALLINT *pDesc, SQLLEN * pDesc);

SQLDescribeCol (SQLHSTMT StatementHandle, SQLSMALLINT ColumnNumber,
                SQLCHAR *ColumnName, SQLSMALLINT BufferLength,
                SQLSMALLINT *nameLengthPtr, SQLSMALLINT *DataTypePtr, SQLULEN *ColumnSizePtr,
                SQLSMALLINT *DecimalDigitsPtr, SQLSMALLINT *NullablePtr);
```
Changes in SQL Data Types

The following four SQL types are still supported on 32-bit only; they are not defined for 64-bit compilers. These types are no longer used for any parameters in MDAC 2.7; use of these types will cause compiler failures on 64-bit platforms.

```c
#ifdef WIN32
typedef SQLULEN SQLROWCOUNT;
typedef SQLULEN SQLROWSETSIZE;
typedef SQLULEN SQLTRANSID;
typedef SQLLEN SQLROWOFFSET;
#endif
#ifdef _WIN64
typedef UINT64 SQLSETPOSIROW;
#else
#define SQLSETPOSIROW SQLUSMALLINT
#endif
#ifdef _WIN64
typedef INT64 SQLLEN;
typedef UINT64 SQLULEN;
#endif
```

The definition of SQLSETPOSIROW has changed for both 32-bit and 64-bit compilers:

```c
#ifdef WIN32
typedef UINT64 SQLSETPOSIROW;
#else
#endif
```

The definitions of SQLLEN and SQLULEN have changed for 64-bit compilers:

```c
#ifdef _WIN64
typedef INT64 SQLLEN;
typedef UINT64 SQLULEN;
#else
#endif
```
Although SQL_C_BOOKMARK is deprecated in ODBC 3.0, for 64-bit compilers on 2.0 clients, this value has changed:

```c
#ifdef _WIN64
#define SQL_C_BOOKMARK SQL_C_UBIGINT
#else
#define SQL_C_BOOKMARK SQL_C_ULONG
#endif
```

The BOOKMARK type is defined differently in the newer headers:

```c
typedef SQLULEN BOOKMARK;
```

### Values Returned from ODBC API Calls Through Pointers

The following ODBC function calls take as an input parameter a pointer to a buffer in which data is returned from the driver. The context and meaning of the data returned is determined by other input parameters for the functions. In some cases, these methods may now return 64-bit (8-byte integer) values instead of the typical 32-bit (4-byte) integer values. These cases are as follows:

#### SQLColAttribute

When the `FieldIdentifier` parameter has one of the following values, a 64-bit value is returned in `*NumericAttribute`:

- SQL_DESC_AUTO_UNIQUE_VALUE
- SQL_DESC_CASE_SENSITIVE
- SQL_DESC_CONCISE_TYPE
- SQL_DESC_COUNT
- SQL_DESC_DISPLAY_SIZE
- SQL_DESC_FIXED_PREC_SCALE
- SQL_DESC_LENGTH
- SQL_DESC_NULLABLE
- SQL_DESC_NUM_PREC_RADIX
- SQL_DESC_OCTET_LENGTH
- SQL_DESC_PRECISION
- SQL_DESC_SCALE
- SQL_DESC_SEARCHABLE
- SQL_DESC_TYPE
- SQL_DESC_UNNAMED
- SQL_DESC_UNSIGNED
- SQL_DESC_UPDATABLE

#### SQLColAttributes

When the `fDescType` parameter has one of the following values, a 64-bit value is returned in `*pDesc`:

- SQL_COLUMN_COUNT
- SQL_COLUMN_DISPLAY_SIZE
- SQL_COLUMN_LENGTH
- SQL_DESC_AUTO_UNIQUE_VALUE
- SQL_DESC_CASE_SENSITIVE
- SQL_DESC_CONCISE_TYPE
- SQL_DESC_FIXED_PREC_SCALE
SQLGetConnectAttr
When the Attribute parameter has one of the following values, a 64-bit value is returned in Value:
- SQL_ATTR_ASYNC_ENABLE
- SQL_ATTR_ENLIST_IN_DTC
- SQL_ATTR_ODBC_CURSORS
- SQL_ATTR_QUIET_MODE

SQLGetConnectOption
When the Attribute parameter has one of the following values, a 64-bit value is returned in Value:
- SQL_ATTR_QUIET_MODE

SQLGetDescField
When the FieldIdentifier parameter has one of the following values, a 64-bit value is returned in *ValuePtr:
- SQL_DESC_ARRAY_SIZE
- SQL_DESC_ARRAY_STATUS_PTR
- SQL_DESC_BIND_OFFSET_PTR
- SQL_DESC_DATA_PTR
- SQL_DESC_DISPLAY_SIZE
- SQL_DESC_INDICATOR_PTR
- SQL_DESC_LENGTH
- SQL_DESC_OCTET_LENGTH
- SQL_DESC_OCTET_LENGTH_PTR
- SQL_DESC_ROWS_PROCESSED_PTR

SQLGetDiagField
When the DiagIdentifier parameter has one of the following values, a 64-bit value is returned in *DiagInfoPtr:
- SQL_DIAG_CURSOR_ROW_COUNT
- SQL_DIAG_ROW_COUNT
- SQL_DIAG_ROW_NUMBER

SQLGetInfo
When the InfoType parameter has one of the following values, a 64-bit value is returned in *InfoValuePtr:
- SQL_DRIVER_HDBC
- SQL_DRIVER_HENV
- SQL_DRIVER_HLIB
When InfoType has either of the following 2 values *InfoValuePtr is 64-bits on both input and output:
- SQL_DRIVER_HDESC
- SQL_DRIVER_HSTMT

SQLGetStmtAttr
When the Attribute parameter has one of the following values, a 64-bit value is returned in *ValuePtr:
- SQL_ATTR_APP_PARAM_DESC
- SQL_ATTR_APP_ROW_DESC
- SQL_ATTR_ASYNC_ENABLE
- SQL_ATTR_CONCURRENCY
- SQL_ATTR_CURSOR_SCROLLABLE
SQL_ATTR_CURSOR_SENSITIVITY
SQL_ATTR_CURSOR_TYPE
SQL_ATTR_ENABLE_AUTO_IPD
SQL_ATTR_FETCH_BOOKMARK_PTR
SQL_ATTR_ROWS_Fetched_PTR
SQL_ATTR_IMP_PARAM_DESC
SQL_ATTR_IMP_ROW_DESC
SQL_ATTR_KEYSET_SIZE
SQL_ATTR_MAX_LENGTH
SQL_ATTR_MAX_ROWS
SQL_ATTR_METADATA_ID
SQL_ATTR_NOSCAN
SQL_ATTR_PARAM_BIND_OFFSET_PTR
SQL_ATTR_PARAM_BIND_TYPE
SQL_ATTR_PARAM_OPERATION_PTR
SQL_ATTR_PARAM_STATUS_PTR
SQL_ATTR_PARAMS_PROCESSED_PTR
SQL_ATTR_PARAMSET_SIZE
SQL_ATTR_QUERY_TIMEOUT
SQL_ATTR_RETRIEVE_DATA
SQL_ATTR_ROW_ARRAY_SIZE
SQL_ATTR_ROW_BIND_OFFSET_PTR
SQL_ATTR_ROW_NUMBER
SQL_ATTR_ROW_OPERATION_PTR
SQL_ATTR_ROW_STATUS_PTR
SQL_ATTR_SIMULATE_CURSOR
SQL_ATTR_USE_BOOKMARKS

**SQLGetStmtOption**

When the `Option` parameter has one of the following values, a 64-bit value is returned in `*Value`:

SQL_KEYSET_SIZE
SQL_MAX_LENGTH
SQL_MAX_ROWS
SQL_ROWSET_SIZE

**SQLSetConnectAttr**

When the `Attribute` parameter has one of the following values, a 64-bit value is passed in `Value`:

SQL_ATTR_ASYNC_ENABLE
SQL_ATTR_ENLIST_IN_DTC
SQL_ATTR_ODBC_CURSORS
SQL_ATTR_QUIET_MODE

**SQLSetConnectOption**

When the `Attribute` parameter has one of the following values, a 64-bit value is passed in `Value`:

SQL_ATTR_QUIET_MODE

**SQLSetDescField**

When the `FieldIdentifier` parameter has one of the following values, a 64-bit value is passed in `*ValuePtr`:
SQLSetStmtAttr

When the Attribute parameter has one of the following values, a 64-bit value is passed in *ValuePtr:

- SQL_ATTR_APP_PARAM_DESC
- SQL_ATTR_APP_ROW_DESC
- SQL_ATTR_ASYNC_ENABLE
- SQL_ATTR_CONCURRENCY
- SQL_ATTR_CURSOR_SCROLLABLE
- SQL_ATTR_CURSOR_SENSITIVITY
- SQL_ATTR_CURSOR_TYPE
- SQL_ATTR_ENABLE_AUTO_IPD
- SQL_ATTR_FETCH_BOOKMARK_PTR
- SQL_ATTR_IMP_PARAM_DESC
- SQL_ATTR_IMP_ROW_DESC
- SQL_ATTR_KEYSET_SIZE
- SQL_ATTR_MAX_LENGTH
- SQL_ATTR_MAX_ROWS
- SQL_ATTR_METADATA_ID
- SQL_ATTR_NOSCAN
- SQL_ATTR_PARAM_BIND_OFFSET_PTR
- SQL_ATTR_PARAM_BIND_TYPE
- SQL_ATTR_PARAM_OPERATION_PTR
- SQL_ATTR_PARAM_STATUS_PTR
- SQL_ATTR_PARAMS_PROCESSED_PTR
- SQL_ATTR_PARAMSET_SIZE
- SQL_ATTR_QUERY_TIMEOUT
- SQL_ATTR_RETRIEVE_DATA
- SQL_ATTR_ROW_ARRAY_SIZE
- SQL_ATTR_ROW_BIND_OFFSET_PTR
- SQL_ATTR_ROW_NUMBER
- SQL_ATTR_ROW_OPERATION_PTR
- SQL_ATTR_ROW_STATUS_PTR
- SQL_ATTR_ROWS_FETCHED_PTR
- SQL_ATTR_SIMULATE_CURSOR
- SQL_ATTR_USE_BOOKMARKS
**SQLSetStmtOption**

When the `Option` parameter has one of the following values, a 64-bit value is passed in `*Value`:

- `SQL_KEYSET_SIZE`
- `SQL_MAX_LENGTH`
- `SQL_MAX_ROWS`
- `SQL_ROWSET_SIZE`

**See Also**

Introduction to ODBC

© 2016 Microsoft

**Developing Applications**

This section contains information about developing applications that use the ODBC interface and drivers that implement it.

This section contains the following topics.

- **ODBC Fundamentals**
- **Basic ODBC Application Steps**
- **Connecting to a Data Source or Driver**
- **Catalog Functions**
- **SQL Statements**
- **Executing Statements**
- **Retrieving Results (Basic)**
- **Retrieving Results (Advanced)**
- **Updating Data Overview**
- **Descriptors**
- **Transactions**
- **Diagnostics**
- **Interoperability**
- **Programming Considerations**

**See Also**

ODBC Programmer's Reference

© 2016 Microsoft

**ODBC Fundamentals**

This section contains the following topics.

- **Handles**
- **Buffers**
- **Data Types in ODBC**
- **Conformance Levels**
- **Environment, Connection, and Statement Attributes**
Handles

Handles are opaque, 32-bit values that identify a particular item; in ODBC, this item can be an environment, connection, statement, or descriptor. When the application calls `SQLAllocHandle`, the Driver Manager or driver creates a new item of the specified type and returns its handle to the application. The application later uses the handle to identify that item when calling ODBC functions. The Driver Manager and driver use the handle to locate information about the item.

For example, the following code uses two statement handles (`hstmtOrder` and `hstmtLine`) to identify the statements on which to create result sets of sales orders and sales order line numbers. It later uses these handles to identify which result set to fetch data from.

```
SQLHSTMT hstmtOrder, hstmtLine; // Statement handles.
SQLUINTEGER OrderID;
SQLINTEGER OrderIDInd = 0;
SQLRETURN rc;

// Prepare the statement that retrieves line number information.
SQLPrepare(hstmtline, "SELECT * FROM Lines WHERE OrderID = ?, SQL_NTS);

// Bind OrderID to the parameter in the preceding statement.
SQLBindParameter(hstmtLine, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0, &OrderID, 0, &OrderIDInd);

// Bind the result sets for the Order table and the Lines table. Bind
// OrderID to the OrderID column in the Orders table. When each row is
// fetched, OrderID will contain the current order ID, which will then be
// passed as a parameter to the statement to fetch line number
// information. Code not shown.

// Create a result set of sales orders.
SQLExecDirect(hstmtOrder, "SELECT * FROM Orders", SQL_NTS);

// Fetch and display the sales order data. Code to check if rc equals
// SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
while ((rc = SQLFetch(hstmtOrder)) != SQL_NO_DATA) {
    // Display the sales order data. Code not shown.
}

// Create a result set of line numbers for the current sales order.
SQLExecute(hstmtLine);

// Fetch and display the sales order line number data. Code to check
// if rc equals SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
while ((rc = SQLFetch(hstmtLine)) != SQL_NO_DATA) {
    // Display the sales order line number data. Code not shown.
}

// Close the sales order line number result set.
SQLCloseCursor(hstmtLine);

// Close the sales order result set.
SQLCloseCursor(hstmtOrder);
```

Handles are meaningful only to the ODBC component that created them; that is, only the Driver Manager can interpret Driver Manager handles and only a driver can interpret its own handles.

For example, suppose the driver in the preceding example allocates a structure to store information about a statement and returns the pointer to this structure as the statement handle. When the application calls `SQLPrepare`, it passes an SQL statement and the handle of the statement used for sales order line numbers. The driver sends the SQL statement to the data source, which prepares it and returns an access plan identifier. The driver uses the handle to find the structure in which to store this identifier.

Later, when the application calls `SQLExecute` to generate the result set of line numbers for a particular sales order, it passes the same handle. The driver uses the handle to retrieve the access plan identifier from the structure. It sends the identifier to the data source to tell it which plan to execute.

ODBC has two levels of handles: Driver Manager handles and driver handles. The application uses Driver Manager handles when calling ODBC functions because it calls those functions in the Driver Manager. The Driver Manager uses this handle to find the corresponding driver handle and uses the driver handle when calling the function in the driver. For an example of how driver and Driver Manager handles are used, see Driver Manager’s Role in the Connection Process.

That there are two levels of handles is an artifact of the ODBC architecture; in most cases, it is not relevant to either the application or driver. Although there is usually no reason to do so, it is possible for the application to determine the driver handles by calling `SQLGetInfo`.

This section contains the following topics.

- Tables and Views
Environment Handles

An environment is a global context in which to access data; associated with an environment is any information that is global in nature, such as:

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

Within a piece of code that implements ODBC (the Driver Manager or a driver), an environment handle identifies a structure to contain this information.

Environment handles are not frequently used in ODBC applications. They are always used in calls to SQLDataSources and SQLDrivers and sometimes used in calls to SQLAllocHandle, SQLEndTran, SQLFreeHandle, SQLGetDiagField, and SQLGetDiagRec.

Each piece of code that implements ODBC (the Driver Manager or a driver) contains one or more environment handles. For example, the Driver Manager maintains a separate environment handle for each application that is connected to it. Environment handles are allocated with SQLAllocHandle and freed with SQLFreeHandle.

Connection Handles

A connection consists of a driver and a data source. A connection handle identifies each connection. The connection handle defines not only which driver to use but which data source to use with that driver. Within a segment of code that implements ODBC (the Driver Manager or a driver), the connection handle identifies a structure that contains connection information, such as the following:

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

ODBC does not prevent multiple simultaneous connections, if the driver supports them. Therefore, in a particular ODBC environment, multiple connection handles might point to a variety of drivers and data sources, to the same driver and a variety of data sources, or even to multiple connections to the same driver and data source. Some drivers limit the number of active connections they support; the SQL_MAX_DRIVER_CONNECTIONS option in SQLGetInfo specifies how many active connections a particular driver supports.

Connection handles are primarily used when connecting to the data source (SQLConnect, SQLDriverConnect, or SQLBrowseConnect), disconnecting from the data source (SQLDisconnect), getting information about the driver and data source (SQLGetInfo), retrieving diagnostics (SQLGetDiagField and SQLGetDiagRec), and performing transactions (SQLEndTran). They are also used when setting and getting connection attributes (SQLSetConnectAttr and SQLGetConnectAttr) and when getting the native format of an SQL statement (SQLNativeSql).

Connection handles are allocated with SQLAllocHandle and freed with SQLFreeHandle.

Statement Handles

A statement is most easily thought of as an SQL statement, such as SELECT * FROM Employee. However, a statement is more than just an SQL statement — it consists of all of the information associated with that SQL statement, such as any result sets created by the statement and parameters used in the execution of the statement. A statement does not even need to have an application-defined SQL statement. For example, when a catalog function such as SQLTables is executed on a statement, it executes a predefined SQL statement that returns a list of table names.

Each statement is identified by a statement handle. A statement is associated with a single connection, and there can be multiple statements on that connection. Some drivers limit the number of active statements they support; the SQL_MAX_CONCURRENT_ACTIVITIES option in SQLGetInfo specifies how
many active statements a driver supports on a single connection. A statement is defined to be active if it has results pending, where results are either a result set or the count of rows affected by an INSERT, UPDATE, or DELETE statement, or data is being sent with multiple calls to SQLPutData.

Within a piece of code that implements ODBC (the Driver Manager or a driver), the statement handle identifies a structure that contains statement information, such as:

- The statement's state
- The current statement-level diagnostics
- The addresses of the application variables bound to the statement's parameters and result set columns
- The current settings of each statement attribute

Statement handles are used in most ODBC functions. Notably, they are used in the functions to bind parameters and result set columns (SQLBindParameter and SQLBindCol), prepare and execute statements (SQLPrepare, SQLExecute, and SQLExecDirect), retrieve metadata (SQLColAttribute and SQLDescribeCol), fetch results (SQLFetch), and retrieve diagnostics (SQLGetDiagField and SQLGetDiagRec). They are also used in catalog functions (SQLColumns, SQLTables, and so on) and a number of other functions.

Statement handles are allocated with SQLAllocHandle and freed with SQLFreeHandle.

Descriptor Handles

A descriptor is a collection of metadata that describes the parameters of an SQL statement or the columns of a result set, as seen by the application or driver (also known as the implementation). Thus, a descriptor can fill any of four roles:

- **Application Parameter Descriptor (APD).** Contains information about the application buffers bound to the parameters in an SQL statement, such as their addresses, lengths, and C data types.
- **Implementation Parameter Descriptor (IPD).** Contains information about the parameters in an SQL statement, such as their SQL data types, lengths, and nullability.
- **Application Row Descriptor (ARD).** Contains information about the application buffers bound to the columns in a result set, such as their addresses, lengths, and C data types.
- **Implementation Row Descriptor (IRD).** Contains information about the columns in a result set, such as their SQL data types, lengths, and nullability.

Four descriptors (one filling each role) are allocated automatically when a statement is allocated. These are known as automatically allocated descriptors and are always associated with that statement. Applications can also allocate descriptors with SQLAllocHandle. These are known as explicitly allocated descriptors. They are allocated on a connection and can be associated with one or more statements on that connection to fulfill the role of an APD or ARD on those statements.

Most operations in ODBC can be performed without explicit use of descriptors by the application. However, descriptors provide a convenient shortcut for some operations. For example, suppose an application wants to insert data from two different sets of buffers. To use the first set of buffers, it would repeatedly call SQLBindParameter to bind them to the parameters in an INSERT statement and then execute the statement. To use the second set of buffers, it would repeat this process. Alternatively, it could set up bindings to the first set of buffers in one descriptor and to the second set of buffers in another descriptor. To switch between the sets of bindings, the application would simply call SQLSetStmtAttr and associate the correct descriptor with the statement as the APD.

For more information about descriptors, see Types of Descriptors.

State Transitions

ODBC defines discrete states for each environment, each connection, and each statement. For example, the environment has three possible states: Unallocated (in which no environment is allocated), Allocated (in which an environment is allocated but no connections are allocated), and Connection (in which an environment and one or more connections are allocated). Connections have seven possible states; statements have 13 possible states.

A particular item, as identified by its handle, moves from one state to another when the application calls a certain function or functions and passes the handle to that item. Such movement is called a state transition. For example, allocating an environment handle with SQLAllocHandle moves the environment from Unallocated to Allocated, and freeing that handle with SQLFreeHandle returns it from Allocated to Unallocated. ODBC defines a limited number of legal state transitions, which is another way of saying that functions must be called in a certain order.

Some functions, such as SQLGetConnectAttr, do not affect state at all. Other functions affect the state of a single item. For example, SQLDisconnect moves a connection from a Connection state to an Allocated state. Finally, some functions affect the state of more than one item. For example, allocating a connection handle with SQLAllocHandle moves a connection from an Unallocated to an Allocated state and moves the environment from an Allocated to a Connection state.

If an application calls a function out of order, the function returns a state transition error. For example, if an environment is in a Connection state and the application calls SQLFreeHandle with that environment handle, SQLFreeHandle returns SQLSTATE HY010 (Function sequence error), because it can be called only when the environment is in an Allocated state. By defining this as an invalid state transition, ODBC prevents the application from freeing the environment while there are active connections.
Some state transitions are inherent in the design of ODBC. For example, it is not possible to allocate a connection handle without first allocating an environment handle, because the function that allocates a connection handle requires an environment handle. Other state transitions are enforced by the Driver Manager and the drivers. For example, SQLExecute executes a prepared statement. If the statement handle passed to it is not in a Prepared state, SQLExecute returns SQLSTATE HY010 (Function sequence error).

From the application's point of view, state transitions are usually straightforward: Legal state transitions tend to go hand-in-hand with the flow of a well-written application. State transitions are more complex for the Driver Manager and the drivers because they must track the state of the environment, each connection, and each statement. Most of this work is done by the Driver Manager; most of the work that must be done by drivers occurs with statements with pending results.

Parts 1 and 2 of this manual ("Introduction to ODBC" and "Developing Applications and Drivers") tend not to explicitly mention state transitions. Instead, they describe the order in which functions must be called. For example, "Executing Statements" states that a statement must be prepared with SQLPrepare before it can be executed with SQLExecute. For a complete description of states and state transitions, including which transitions are checked by the Driver Manager and which must be checked by drivers, see Appendix B: ODBC State Transition Tables.

© 2016 Microsoft

Buffers

A buffer is any piece of application memory used to pass data between the application and the driver. For example, application buffers can be associated with, or bound to, result set columns with SQLBindCol. As each row is fetched, the data is returned for each column in these buffers. Input buffers are used to pass data from the application to the driver; output buffers are used to return data from the driver to the application.

Note

If an ODBC function returns SQL_ERROR, the contents of any output arguments to that function are undefined.

This discussion concerns itself primarily with buffers of indeterminate type. The addresses of these buffers appear as arguments of type SQLPOINTER, such as the TargetValuePtr argument in SQLBindCol. However, some of the items discussed here, such as the arguments used with buffers, also apply to arguments used to pass strings to the driver, such as the TableName argument in SQLTables.

These buffers usually come in pairs. Data buffers are used to pass the data itself, while length/indicator buffers are used to pass the length of the data in the data buffer or a special value such as SQL_NULL_DATA, which indicates that the data is NULL. The length of the data in a data buffer is different from the length of the data buffer itself. The following illustration shows the relationship between the data buffer and the length/indicator buffer.

Data Buffer

* * *

Length/Indicator Buffer

* * *

Buffer

Data

Data length

A length/indicator buffer is required whenever the data buffer contains variable-length data, such as character or binary data. If the data buffer contains fixed-length data, such as an integer or date structure, a length/indicator buffer is needed only to pass indicator values because the length of the data is already known. If an application uses a length/indicator buffer with fixed-length data, the driver ignores any lengths passed in it.

The length of both the data buffer and the data it contains is measured in bytes, as opposed to characters. This distinction is unimportant for programs that use ANSI strings because lengths in bytes and characters are the same.

When the data buffer represents a driver-defined descriptor field, diagnostic field, or attribute, the application should indicate to the Driver Manager the nature of the function argument that indicates the value for the field or attribute. The application does this by setting the length argument in any function call that sets the field or attribute to one of the following values. (The same is true for functions that retrieve the values of the field or attribute, with the exception that the argument points to the values that for the setting function are in the argument itself.)

- If the function argument that indicates the value for the field or attribute is a pointer to a character string, the length argument is the length of the string or SQL_NTS.
- If the function argument that indicates the value for the field or attribute is a pointer to a binary buffer, the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in the length argument. This places a negative value in the length argument.
- If the function argument that indicates the value for the field or attribute is a pointer to a value other than a character string or a binary string, the length argument should have the value SQL_IS_POINTER.
- If the function argument that indicates the value for the field or attribute contains a fixed-length value, the length argument is SQL_IS_INTEGER, SQL_IS_SHORT, SQL_IS_SMALLINT, or SQL_ISI_USMALLINT, as appropriate.

This section contains the following topics.

- Deferred Buffers
Deferred Buffers

A deferred buffer is one whose value is used at some time after it is specified in a function call. For example, `SQLBindParameter` is used to associate, or bind, a data buffer with a parameter in an SQL statement. The application specifies the number of the parameter and passes the address, byte length, and type of the buffer. The driver saves this information but does not examine the contents of the buffer. Later, when the application executes the statement, the driver retrieves the information and uses it to retrieve the parameter data and send it to the data source. Therefore, the input of data in the buffer is deferred. Because deferred buffers are specified in one function and used in another, it is an application programming error to free a deferred buffer while the driver still expects it to exist; for more information, see Allocating and Freeing Buffers, later in this section.

Both input and output buffers can be deferred. The following table summarizes the uses of deferred buffers. Note that deferred buffers bound to result set columns are specified with `SQLBindCol`, and deferred buffers bound to SQL statement parameters are specified with `SQLBindParameter`.

<table>
<thead>
<tr>
<th>Buffer use</th>
<th>Type</th>
<th>Specified with</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending data for input parameters</td>
<td>Deferred input</td>
<td><code>SQLBindParameter</code></td>
<td><code>SQLExecute</code> <code>SQLExecDirect</code></td>
</tr>
<tr>
<td>Sending data to update or insert a new row in a result set</td>
<td>Deferred input</td>
<td><code>SQLBindCol</code></td>
<td><code>SQLSetPos</code></td>
</tr>
<tr>
<td>Returning data for output and input/output parameters</td>
<td>Deferred output</td>
<td><code>SQLBindParameter</code></td>
<td><code>SQLExecute</code> <code>SQLExecDirect</code></td>
</tr>
<tr>
<td>Returning result set data</td>
<td>Deferred output</td>
<td><code>SQLBindCol</code></td>
<td><code>SQLFetch</code> <code>SQLFetchScroll</code> <code>SQLSetPos</code></td>
</tr>
</tbody>
</table>

Allocating and Freeing Buffers

All buffers are allocated and freed by the application. If a buffer is not deferred, it need only exist for the duration of the call to a function. For example, `SQLGetInfo` returns the value associated with a particular option in the buffer pointed to by the `InfoValuePtr` argument. This buffer can be freed immediately after the call to `SQLGetInfo`, as shown in the following code example:

```sql
SQLSMALLINT InfoValueLen;
SQLCHAR * InfoValuePtr = malloc(50); // Allocate InfoValuePtr.
SQLGetInfo(hdbc, SQL_DBMS_NAME, (SQLPOINTER)InfoValuePtr, 50, &InfoValueLen);
free(InfoValuePtr); // OK to free InfoValuePtr.
```

Because deferred buffers are specified in one function and used in another, it is an application programming error to free a deferred buffer while the driver still expects it to exist. For example, the address of the `ValuePtr` buffer is passed to `SQLBindCol` for later use by `SQLFetch`. This buffer cannot be freed until the column is unbound, such as with a call to `SQLBindCol` or `SQLFreeStmt` as shown in the following code example:

```sql
SQLRETURN rc;
SQLINTEGER ValueLenOrInd;
SQLHSTMT hstmt;

// Allocate ValuePtr
SQLCHAR * ValuePtr = malloc(50);

// Bind ValuePtr to column 1. It is an error to free ValuePtr here.
SQLBindCol(hstmt, 1, SQL_C_CHAR, ValuePtr, 50, &ValueLenOrInd);

// Fetch each row of data and place the value for column 1 in *ValuePtr.
// Code to check if rc equals SQL_ERROR or SQL_SUCCESS_WITH_INFO // not shown.
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
  // It is an error to free ValuePtr here.
}
```
Such an error is easily made by declaring the buffer locally in a function; the buffer is freed when the application leaves the function. For example, the following code causes undefined and probably fatal behavior in the driver:

```c
// Unbind ValuePtr from column 1. It is now OK to free ValuePtr.
SQLFreeStmt(hstmt, SQL_UNBIND);
free(ValuePtr);
```

Using Data Buffers

Data buffers are described by three pieces of information: their type, address, and byte length. Whenever a function needs one of these pieces of information and does not already know it, it has an argument with which the application passes it.

This section contains the following topics.

- Data Buffer Type
- Data Buffer Address
- Data Buffer Length

Data Buffer Type

The C data type of a buffer is specified by the application. With a single variable, this occurs when the application allocates the variable. With generic memory — that is, memory pointed to by a pointer of type void — this occurs when the application casts the memory to a particular type. The driver discovers this type in two ways:

- **Data buffer type argument.** Buffers used to transfer parameter values and result set data, such as the buffer bound with `TargetValuePtr` in `SQLBindCol`, usually have an associated type argument, such as the `TargetType` argument in `SQLBindCol`. In this argument, the application passes the C type identifier that corresponds to the type of the buffer. For example, in the following call to `SQLBindCol`, the value `SQL_C_TYPE_DATE` tells the driver that the `Date` buffer is an `SQL_DATE_STRUCT`:

  ```c
  SQL_DATE_STRUCT Date;
  SQLINTEGER DateInd;
  SQLBindCol(hstmt, 1, SQL_C_TYPE_DATE, &Date, 0, &DateInd);
  ```

For more information about type identifiers, see the Data Types in ODBC section, later in this section.
Predefined type. Buffers used to send and retrieve options or attributes, such as the buffer pointed to by the `InfoValuePtr` argument in `SQLGetInfo`, have a fixed type that depends on the option specified. The driver assumes that the data buffer is of this type; it is the application's responsibility to allocate a buffer of this type. For example, in the following call to `SQLGetInfo`, the driver assumes the buffer is a 32-bit integer because this is what the `SQL_STRING_FUNCTIONS` option requires:

```c
SQLUINTEGER StringFuncs;
SQLGetInfo(hdbc, SQL_STRING_FUNCTIONS, (SQLPOINTER) &StringFuncs, 0, NULL);
```

The driver uses the C data type to interpret the data in the buffer.

### Data Buffer Address

The application passes the address of the data buffer to the driver in an argument, often named `ValuePtr` or a similar name. For example, in the following call to `SQLBindCol`, the application specifies the address of the `Date` variable:

```c
SQL_DATE_STRUCT Date;
SQLINTEGER DateInd;
SQLBindCol(hstmt, 1, SQL_C_TYPE_DATE, &dsDate, 0, &DateInd);
```

As mentioned in the Allocating and Freeing Buffers section, the address of a deferred buffer must remain valid until the buffer is unbound.

Unless it is specifically prohibited, the address of a data buffer can be a null pointer. For buffers used to send data to the driver, this causes the driver to ignore the information normally contained in the buffer. For buffers used to retrieve data from the driver, this causes the driver to not return a value. In both cases, the driver ignores the corresponding data buffer length argument.

### Data Buffer Length

The application passes the byte length of the data buffer to the driver in an argument, named `BufferLength` or a similar name. For example, in the following call to `SQLBindCol`, the application specifies the length of the `ValuePtr` buffer (`sizeof(ValuePtr)`):

```c
SQLCHAR ValuePtr[50];
SQLINTEGER ValueLenOrInd;
SQLBindCol(hstmt, 1, SQL_C_CHAR, ValuePtr, sizeof(ValuePtr), &ValueLenOrInd);
```

A driver will always return the number of bytes, not the number of characters, in the buffer length argument of any function that has an output string argument.

Data buffer lengths are required only for output buffers; the driver uses them to avoid writing past the end of the buffer. However, the driver checks the data buffer length only when the buffer contains variable-length data, such as character or binary data. If the buffer contains fixed-length data, such as an integer or date structure, the driver ignores the data buffer length and assumes the buffer is large enough to hold the data; that is, it never truncates fixed-length data. It is therefore important for the application to allocate a large enough buffer for fixed-length data.

When a truncation of non-data output strings occurs (such as the cursor name returned for `SQLGetCursorName`), the returned length in the buffer length argument is the maximum character length possible.

Data buffer lengths are not required for input buffers because the driver does not write to these buffers.

This section contains the following topics.

- Using Length/Indicator Values
- Data Length, Buffer Length, and Truncation
- Character Data and C Strings

### Using Length/Indicator Values
The length/indicator buffer is used to pass the byte length of the data in the data buffer or a special indicator such as SQL_NULL_DATA, which indicates that the data is NULL. Depending on the function in which it is used, a length/indicator buffer is defined to be an SQLINTEGER or an SQLSMALLINT. Therefore, a single argument is needed to describe it. If the data buffer is a nondeferred input buffer, this argument contains the byte length of the data itself or an indicator value. It is often named StrLen_or_Ind or a similar name. For example, the following code calls SQLPutData to pass a buffer full of data; the byte length (ValueLen) is passed directly because the data buffer (ValuePtr) is an input buffer.

```c
SQLCHAR ValuePtr[50];
SQLINTEGER ValueLen;

// Call local function to place data in ValuePtr. In ValueLen, return the
// number of bytes of data placed in ValuePtr. If there is not enough
// data, this will be less than 50.
FillBuffer(ValuePtr, sizeof(ValuePtr), &ValueLen);

// Call SQLPutData to send the data to the driver.
SQLPutData(hstmt, ValuePtr, ValueLen);
```

If the data buffer is a deferred input buffer, a nondeferred output buffer, or an output buffer, the argument contains the address of the length/indicator buffer. It is often named StrLen_or_IndPtr or a similar name. For example, the following code calls SQLGetData to retrieve a buffer full of data; the byte length is returned to the application in the length/indicator buffer (ValueLenOrInd), whose address is passed to SQLGetData because the corresponding data buffer (ValuePtr) is a nondeferred output buffer.

```c
SQLCHAR ValuePtr[50];
SQLINTEGER ValueLenOrInd;

SQLGetData(hstmt, 1, SQL_C_CHAR, ValuePtr, sizeof(ValuePtr), &ValueLenOrInd);
```

Unless it is specifically prohibited, a length/indicator buffer argument can be 0 (if nondeferred input) or a null pointer (if output or deferred input). For input buffers, this causes the driver to ignore the byte length of the data. This returns an error when passing variable-length data but is common when passing non-null, fixed-length data, because neither a length nor an indicator value is needed. For output buffers, this causes the driver to not return the byte length of the data or an indicator value. This is an error if the data returned by the driver is NULL but is common when retrieving fixed-length, non-nullable data, because neither a length nor an indicator value is needed.

As when the address of a deferred data buffer is passed to the driver, the address of a deferred length/indicator buffer must remain valid until the buffer is unbound.

The following lengths are valid as length/indicator values:

- \( n \), where \( n > 0 \).
- 0.
- SQL_NTS. A string sent to the driver in the corresponding data buffer is null-terminated; this is a convenient way for C programmers to pass strings without having to calculate their byte length. This value is legal only when the application sends data to the driver. When the driver returns data to the application, it always returns the actual byte length of the data.

The following values are valid as length/indicator values. SQL_NULL_DATA is stored in the SQL_DESC_INDICATOR_PTR descriptor field; all other values are stored in the SQL_DESC_OCTET_LENGTH_PTR descriptor field.

- SQL_NULL_DATA. The data is a NULL data value, and the value in the corresponding data buffer is ignored. This value is legal only for SQL data sent to or retrieved from the driver.
- SQL_DATA_AT_EXEC. The data buffer does not contain any data. Instead, the data will be sent with SQLPutData when the statement is executed or when SQLBulkOperations or SQLSetPos is called. This value is legal only for SQL data sent to the driver. For more information, see SQLBindParameter, SQLBulkOperations, and SQLSetPos.
- Result of the SQL_LEN_DATA_AT_EXEC(length) macro. This value is similar to SQL_DATA_AT_EXEC. For more information, see Sending Long Data.
- SQL_NO_TOTAL. The driver cannot determine the number of bytes of long data still available to return in an output buffer. This value is legal only for SQL data retrieved from the driver.
- SQL_DEFAULT_PARAM. A procedure is to use the default value of an input parameter in a procedure instead of the value in the corresponding data buffer.
- SQL_COLUMN_IGNORE. SQLBulkOperations or SQLSetPos is to ignore the value in the data buffer. When updating a row of data by a call to SQLBulkOperations or SQLSetPos, the column value is not changed. When inserting a new row of data by a call to SQLBulkOperations, the column value is set to its default or, if the column does not have a default, to NULL.

Data Length, Buffer Length, and Truncation

The data length is the byte length of the data as it would be stored in the application’s data buffer; not as it is stored in the data source. This distinction is important because the data is often stored in different types in the data buffer than in the data source. So for data being sent to the data source, this is the...
Character Data and C Strings

Input parameters that refer to variable-length character data (such as column names, dynamic parameters, and string attribute values) have an associated length parameter. If the application terminates strings with the null character, as is typical in C, it provides as an argument either the length in bytes of the string (not including the null-terminator) or SQL_NTS (Null-Terminated String). A non-negative length argument specifies the actual length of the associated string. The length argument may be 0 to specify a zero-length string, which is distinct from a NULL value. The negative value SQL_NTS directs the driver to determine the length of the string by locating the null-termination character.

When character data is returned from the driver to the application, the driver must always null-terminate it. This gives the application the choice of whether to handle the data as a string or a character array. If the application buffer is not large enough to return all of the character data, the driver truncates it to the byte length of the buffer less the number of bytes required by the null-termination character, null-terminates the truncated data, and stores it in the buffer. Therefore, applications must always allocate extra space for the null-termination character in buffers used to retrieve character data. For example, a 51-byte buffer is needed to retrieve 50 characters of data.

Special care must be taken by both the application and the driver when sending or retrieving long character data in parts with SQLPutData or SQLGetData. If the data is passed as a series of null-terminated strings, the null-termination characters on these strings must be stripped before the data can be reassembled.

A number of ODBC programmers have confused character data and C strings. That this has occurred is an artifact of using the C language when defining ODBC functions. If an ODBC driver or application uses another language — remember that ODBC is language-independent — this confusion is less likely to arise.

When C strings are used to hold character data, the null-termination character is not considered to be part of the data and is not counted as part of its byte length. For example, the character data "ABC" can be held as the C string "ABC\0" or the character array ('A', 'B', 'C'). The byte length of the data is 3, whether or not it is treated as a string or a character array.

Although applications and drivers commonly use C strings (null-terminated arrays of characters) to hold character data, there is no requirement to do this. In C, character data can also be treated as an array of characters (without null-termination) and its byte length passed separately in the length/indicator buffer.

Because character data can be held in a non-null-terminated array and its byte length passed separately, it is possible to embed null characters in character data. However, the behavior of ODBC functions in this case is undefined and it is driver-specific whether a driver handles this correctly. Thus, interoperable applications should always handle character data that can contain embedded null characters as binary data.
Type Identifiers

To describe SQL and C data types, ODBC defines two sets of type identifiers. A type identifier describes the type of an SQL column or a C buffer. It is a `#define` value and is generally passed as a function argument or returned in metadata.

For example, the following call to `SQLBindParameter` binds a variable of type `SQL_DATE_STRUCT` to a date parameter in an SQL statement. The C type identifier `SQL_C_TYPE_DATE` specifies the type of the Date variable, and the SQL type identifier `SQL_TYPE_DATE` specifies the type of the dynamic parameter.

```c
SQL_DATE_STRUCT Date;
SQLINTEGER DateInd = 0;
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_TYPE_DATE, SQL_TYPE_DATE, 0, 0, &Date, 0, &DateInd);
```

SQL Data Types in ODBC

SQL data types are the types in which data is stored in the data source.

This section contains the following topics.

- SQL Type Identifiers
- Retrieving Data Type Information with `SQLGetTypeInfo`

SQL Type Identifiers

Each data source defines its own SQL data types. ODBC defines type identifiers and describes the general characteristics of the SQL data types that might be mapped to each type identifier. It is driver-specific how each data type in the underlying data source is mapped to an SQL type identifier of ODBC.

For example, `SQL_CHAR` is the type identifier for a character column with a fixed length, typically between 1 and 254 characters. These characteristics correspond to the CHAR data type found in many SQL data sources. Thus, when an application discovers that the type identifier for a column is `SQL_CHAR`, it can assume it is probably dealing with a CHAR column. However, it should still check the byte length of the column before assuming it is between 1 and 254 characters; the driver for a non-SQL data source, for example, might map a fixed-length character column of 500 characters to `SQL_CHAR` or `SQL_LONGVARCHAR`, because neither is an exact match.

ODBC defines a wide variety of SQL type identifiers. However, the driver is not required to use all of these identifiers. Instead, it uses only those identifiers it needs to expose the SQL data types supported by the underlying data source. If the underlying data source supports SQL data types to which no type identifier corresponds, the driver can define additional type identifiers. For more information, see Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes.

For a complete description of SQL type identifiers, see C Data Types in Appendix D: Data Types.

Retrieving Data Type Information with `SQLGetTypeInfo`

Because the mappings from underlying SQL data types to ODBC type identifiers are approximate, ODBC provides a function (`SQLGetTypeInfo`) through which a driver can completely describe each SQL data type in the data source. This function returns a result set, each row of which describes the characteristics of a single data type, such as name, type identifier, precision, scale, and nullability.

This information generally is used by generic applications that allow the user to create and alter tables. Such applications call `SQLGetTypeInfo` to retrieve the data type information and then present some or all of it to the user. Such applications need to be aware of two things:

- More than one SQL data type can map to a single type identifier, which can make it difficult to determine which data type to use. To solve this, the result set is ordered first by type identifier and second by closeness to the type identifier's definition. In addition, data source-defined data types take precedence over user-defined data types. For example, suppose that a data source defines the INTEGER and COUNTER data types to be the same except that COUNTER is auto-incrementing. Suppose also that the user-defined type WHOOLENUM is a synonym of INTEGER. Each of these types maps to `SQL_INTEGER`. In the `SQLGetTypeInfo` result set, INTEGER appears first, followed by WHOOLENUM and then COUNTER. WHOOLENUM appears after INTEGER because it is user-defined, but before COUNTER because it more closely matches the definition of the `SQL_INTEGER` type identifier.
ODBC does not define data type names for use in `CREATE TABLE` and `ALTER TABLE` statements. Instead, the application should use the name returned in the `TYPE_NAME` column of the result set returned by `SQLGetTypeInfo`. The reason for this is that although most of SQL does not vary much across DBMSs, data type names vary tremendously. Rather than forcing drivers to parse SQL statements and replace standard data type names with DBMS-specific data type names, ODBC requires applications to use the DBMS-specific names in the first place.

Note that `SQLGetTypeInfo` does not necessarily describe all of the data types an application can encounter. In particular, result sets might contain data types not directly supported by the data source. For example, the data types of the columns in result sets returned by catalog functions are defined by ODBC and these data types might not be supported by the data source. To determine the characteristics of the data types in a result set, an application calls `SQLColAttribute`.

© 2016 Microsoft

C Data Types in ODBC

ODBC defines the C data types that are used by application variables and their corresponding type identifiers. These are used by the buffers that are bound to result set columns and statement parameters. For example, suppose an application wants to retrieve data from a result set column in character format. It declares a variable with the `SQLCHAR *` data type and binds this variable to the result set column with a type identifier of `SQL_C_CHAR`. For a complete list of C data types and type identifiers, see Appendix D: Data Types.

ODBC also defines a default mapping from each SQL data type to a C data type. For example, a 2-byte integer in the data source is mapped to a 2-byte integer in the application. To use the default mapping, an application specifies the `SQL_C_DEFAULT` type identifier. However, use of this identifier is discouraged for interoperability reasons.

All integer C data types defined in ODBC 1.x were signed. Unsigned C data types and their corresponding type identifiers were added in ODBC 2.0. Because of this, applications and drivers need to be particularly careful when dealing with 1.x versions.

C Data Type Extensibility

In ODBC 3.8, you can specify driver-specific C data types. This enables you to bind a SQL type as a driver-specific C type in ODBC applications when you call `SQLBindCol`, `SQLGetData`, or `SQLBindParameter`. This can be useful for supporting new server types, because existing C data types might not correctly represent the new server data types. Using driver-specific C types can increase the number of conversions that drivers can perform.

For example, suppose a database management system (DBMS) introduced a new SQL type, `DATETIMEOFFSET`, to represent the date and time with time zone information. There would be no specific C type in ODBC that corresponded to `DATETIMEOFFSET`. An application would have to bind `DATETIMEOFFSET` as `SQL_C_BINARY` and cast it to a user-defined data type. Beginning in ODBC 3.8 with C data type extensibility, a driver can define a new corresponding C type. For example, for the new SQL type DATETIMEOFFSET, the driver can define a new corresponding C type such as `SQL_C_DATETIMEOFFSET`. Then, an application can bind the new SQL type as a driver-specific C type.

A C data type is defined in the driver as follows:

- The ODBC compliance level for an application, ODBC driver, and Driver Manager is 3.8 (or higher).
- The data range of a driver-specific C type is between 0x4000 and 0x7FFF.
- The driver defines the structure of the data corresponding to the C type. This can be done in the driver-specific SDK.

The driver manager will not validate a C type defined in the range of 0x4000 and 0x7FFF; the driver will perform the validation and any data type conversion. But if the data range of a C type passed to the driver manager is between 0x0000 and 0x3FFF or between 0x8000 and 0xFFFF, the driver manager will validate the C data type.

I Note

Driver-specific C data types should be described in the driver documentation.

To specify an ODBC compliance level of 3.8, an application calls `SQLSetEnvAttr` with the `SQL_ATTR_ODBC_VERSION` attribute set to `SQL_OV_ODBC3_80`. To determine the version of the driver, an application calls `SQLGetInfo` with `SQL_DRIVER_ODBC_VER`.

For more information about ODBC 3.8, see What’s New in ODBC 3.8.

See Also

C Data Types

© 2016 Microsoft

Data Type Conversions

Data can be converted from one type to another at one of four times: when data is transferred from one application variable to another (C to C), when data in an application variable is sent to a statement parameter (C to SQL), when data in a result set column is returned in an application variable (SQL to C), and
when data is transferred from one data source column to another (SQL to SQL).

Any conversion that occurs when data is transferred from one application variable to another is outside the scope of this document.

When an application binds a variable to a result set column or statement parameter, the application implicitly specifies a data type conversion in its choice of the data type of the application variable. For example, suppose a column contains integer data. If the application binds an integer variable to the column, it specifies that no conversion be done; if the application binds a character variable to the column, it specifies that the data be converted from integer to character.

ODBC defines how data is converted between each SQL and C data type. Basically, ODBC supports all reasonable conversions, such as character to integer and integer to float, and does not support ill-defined conversions, such as float to date. Drivers are required to support all conversions for each SQL data type they support. For a complete list of conversions between SQL and C data types, see Converting Data from SQL to C Data Types and Converting Data from C to SQL Data Types in Appendix D: Data Types.

ODBC also defines a scalar function for converting data from one SQL data type to another. The `CONVERT` scalar function is mapped by the driver to the underlying scalar function or functions defined to perform conversions in the data source. Because this function is mapped to DBMS-specific functions, ODBC does not define how these conversions work or what conversions must be supported. An application discovers what conversions are supported by a particular driver and data source through the SQL_CONVERT options in `SQLGetInfo`. For more information about the `CONVERT` scalar function, see Escape Sequences in ODBC and Explicit Data Type Conversion Function.

© 2016 Microsoft

Conformance Levels

ODBC drivers give the application access to diverse data sources. Each driver lets the application determine at run time what ODBC capabilities and what SQL grammar the driver and each data source supports. This is not a requirement of applications designed to work with a single driver or a small, known set of drivers, because these applications can simply be written to the capabilities of that driver or drivers. To help applications discover driver and data source capabilities, two areas of conformance are available: the ODBC interface and SQL grammar.

This section contains the following topics.

- Interface Conformance Levels
- SQL Conformance Levels

© 2016 Microsoft

Interface Conformance Levels

The purpose of leveling is to inform the application what features are available to it from the driver. A leveling scheme based on functions does not sufficiently achieve this goal. In ODBC 3.x, drivers are classified based on the features they possess. Supporting the feature can include supporting the function; it can also include supporting a descriptor field, a statement attribute, a "Y" value for an information type returned by `SQLGetInfo`, and so on.

To simplify specification of interface conformance, ODBC defines three conformance levels. To meet a particular conformance level, a driver must satisfy all of the requirements of that conformance level. Conformance with a given level implies complete conformance with all lower levels.

Conformance levels do not always divide neatly into support for a specific list of ODBC functions, but specify supported features as listed in the following sections. To provide support for a feature, a driver must support some or all forms of calls to certain ODBC functions (for more information, see Function Conformance), setting certain attributes (see Attribute Conformance), and certain descriptor fields (see Descriptor Field Conformance).

The application discovers a driver's interface conformance level by connecting to a data source and calling `SQLGetInfo` with the `SQL_ODBC_INTERFACE_CONFORMANCE` option.

Drivers are free to implement features beyond the level to which they claim complete conformance. Applications discover any such additional capabilities by calling `SQLGetFunctions` (to determine which ODBC functions are present) and `SQLGetInfo` (to query various other ODBC capabilities).

There are three ODBC interface conformance levels: Core, Level 1, and Level 2.

Note

These conformance levels have different requirements than the ODBC API conformance levels of the same name in ODBC 2.x. In particular, all the features implied by ODBC 2.x API conformance Level 1 are now part of the Core interface conformance level. As a result, many ODBC drivers may report Core-level interface conformance.

This section contains the following topics.

- Core Interface Conformance
- Level 1 Interface Conformance
- Level 2 Interface Conformance
- Function Conformance
- Attribute Conformance
Descriptor Field Conformance

Core Interface Conformance

All ODBC drivers must exhibit at least Core-level interface conformance. Because the features in the Core level are those required by most generic interoperable applications, the driver can work with such applications. The features in the Core level also correspond to the features defined in the ISO CLI specification and to the nonoptional features defined in the Open Group CLI specification. A Core-level interface-conformant ODBC driver allows the application to do all of the following:

- Allocate and free all types of handles, by calling SQLAllocHandle and SQLFreeHandle.
- Use all forms of the SQLFreeStmt function.
- Bind result set columns, by calling SQLBindCol.
- Handle dynamic parameters, including arrays of parameters, in the input direction only, by calling SQLBindParameter and SQLNumParams. (Parameters in the output direction are feature 203 in Level 2 Interface Conformance.)
- Specify a bind offset.
- Use the data-at-execution dialog, involving calls to SQLParamData and SQLEndData.
- Manage cursors and cursor names, by calling SQLCloseCursor, SQLGetCursorName, and SQLSetCursorName.
- Gain access to the description (metadata) of result sets, by calling SQLColAttribute, SQLDescribeCol, SQLNumResultCols, and SQLRowCount. (Use of these functions on column number 0 to retrieve bookmark metadata is feature 204 in Level 2 Interface Conformance.)
- Query the data dictionary, by calling the catalog functions SQLColumns, SQLGetTypeInfo, SQLStatistics, and SQLTables.
  
  The driver is not required to support multipart names of database tables and views. However, certain features of the SQL-92 specification, such as column qualification and names of indexes, are syntactically comparable to multipart naming. The present list of ODBC features is not intended to introduce new options into these aspects of SQL-92.
- Manage data sources and connections, by calling SQLConnect, SQLDataSources, SQLDisconnect, and SQLDriverConnect. Obtain information on drivers, no matter which ODBC level they support, by calling SQLDrivers.
- Prepare and execute SQL statements, by calling SQLExecDirect, SQLExecute, and SQLPrepare.
- Fetch one row of a result set or multiple rows, in the forward direction only, by calling SQLFetch or by calling SQLFetchScroll with the FetchOrientation argument set to SQL_FETCH_NEXT.
- Obtain an unbound column in parts, by calling SQLGetData.
- Obtain current values of all attributes, by calling SQLGetConnectAttr, SQLGetEnvAttr, and SQLGetStmtAttr, and set all attributes to their default values and set certain attributes to nondefault values by calling SQLSetConnectAttr, SQLSetEnvAttr, and SQLSetStmtAttr.
- Manipulate certain fields of descriptors, by calling SQLCopyDesc, SQLGetDescField, SQLGetDescRec, SQLSetDescField, and SQLSetDescRec.
- Obtain diagnostic information, by calling SQLGetDiagField and SQLGetDiagRec.
- Detect driver capabilities, by calling SQLGetFunctions and SQLGetInfo. Also, detect the result of any text substitutions made to an SQL statement before it is sent to the data source, by calling SQLNativeSql.
- Use the syntax of SQLEndTran to commit a transaction. A Core-level driver need not support true transactions; therefore, the application cannot specify SQL.Rollback nor SQL_AutoCommit_Off for the SQL_ATTR_AutoCommit connection attribute. (For more information, see feature 109 in Level 2 Interface Conformance.)
- Call SQLCancel to cancel the data-at-execution dialog and, in multithread environments, to cancel an ODBC function executing in another thread. Core-level interface conformance does not mandate support for asynchronous execution of functions, nor the use of SQLCancel to cancel an ODBC function executing asynchronously. Neither the platform nor the ODBC driver need be multithread for the driver to conduct independent activities at the same time. However, in multithread environments, the ODBC driver must be thread-safe. Serialization of requests from the application is a conformant way to implement this specification, even though it might create serious performance problems.
- Obtain the SQL_BEST_ROWID row-identifying column of tables, by calling SQLSpecialColumns. (Support for SQL_ROWVER is feature 208 in Level 2 Interface Conformance.)

Important

ODBC Drivers must implement the functions in the Core interface conformance level.

Level 1 Interface Conformance
The Level 1 interface conformance level includes the Core interface conformance level functionality plus additional features, such as transactions, that are usually available in an OLTP relational DBMS. A Level 1 interface-conformant driver lets the application do the following, in addition to the features in the Core interface conformance level:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Specify the schema of database tables and views (using two-part naming). (For more information, see the three-part naming feature 201 in Level 2 Interface Conformance.)</td>
</tr>
<tr>
<td>102</td>
<td>Invoke true asynchronous execution of ODBC functions, where applicable ODBC functions are all synchronous or all asynchronous on a given connection.</td>
</tr>
<tr>
<td>103</td>
<td>Use scrollable cursors, and thereby achieve access to a result set in methods other than forward-only, by calling <code>SQLFetchScroll</code> with the <code>FetchOrientation</code> argument other than <code>SQL_FETCH_NEXT</code>. (The <code>SQL_FETCH_BOOKMARK FetchOrientation</code> is in feature 204 in Level 2 Interface Conformance.)</td>
</tr>
<tr>
<td>104</td>
<td>Obtain primary keys of tables, by calling <code>SQLPrimaryKeys</code>.</td>
</tr>
<tr>
<td>105</td>
<td>Use stored procedures, through the ODBC escape sequence for procedure calls, and query the data dictionary regarding stored procedures, by calling <code>SQLProcedureColumns</code> and <code>SQLProcedures</code>. (The process by which procedures are created and stored on the data source is outside the scope of this document.)</td>
</tr>
<tr>
<td>106</td>
<td>Connect to a data source by interactively browsing the available servers, by calling <code>SQLBrowseConnect</code>.</td>
</tr>
<tr>
<td>107</td>
<td>Use ODBC functions instead of SQL statements to perform certain database operations: <code>SQLSetPos</code> with <code>SQL_POSITION</code> and <code>SQL_REFRESH</code>.</td>
</tr>
<tr>
<td>108</td>
<td>Gain access to the contents of multiple result sets generated by batches and stored procedures, by calling <code>SQLMoreResults</code>.</td>
</tr>
<tr>
<td>109</td>
<td>Delimit transactions spanning several ODBC functions, with true atomicity and the ability to specify <code>SQL_ROLLBACK</code> in <code>SQLEndTran</code>.</td>
</tr>
</tbody>
</table>

Level 2 Interface Conformance

The Level 2 interface conformance level includes the Level 1 interface conformance-level functionality plus the following features:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>Use three-part names of database tables and views. (For more information, see the two-part naming support feature 101 in Level 1 Interface Conformance.)</td>
</tr>
<tr>
<td>202</td>
<td>Describe dynamic parameters, by calling <code>SQLDescribeParam</code>.</td>
</tr>
<tr>
<td>203</td>
<td>Use not only input parameters but also output and input/output parameters, and result values of stored procedures.</td>
</tr>
<tr>
<td>204</td>
<td>Use bookmarks, including retrieving bookmarks, by calling <code>SQLDescribeCol</code> and <code>SQLColAttribute</code> on column number 0; fetching based on a bookmark, by calling <code>SQLFetchScroll</code> with the <code>FetchOrientation</code> argument set to <code>SQL_FETCH_BOOKMARK</code>; and update, delete, and fetch by bookmark operations, by calling <code>SQLBulkOperations</code> with the <code>Operation</code> argument set to <code>SQL_UPDATE_BY.BookMARK</code>, <code>SQL_DELETE_BY.BookMARK</code>, or <code>SQL_FETCH_BY.BookMARK</code>.</td>
</tr>
<tr>
<td>205</td>
<td>Retrieve advanced information about the data dictionary, by calling <code>SQLColumnPrivileges</code>, <code>SQLForeignKeys</code>, and <code>SQLTablePrivileges</code>.</td>
</tr>
<tr>
<td>206</td>
<td>Use ODBC functions instead of SQL statements to perform additional database operations, by calling <code>SQLBulkOperations</code> with <code>SQL_ADD</code>, or <code>SQLSetPos</code> with <code>SQL.DELETE</code> or <code>SQL.UPDATE</code>. (Support for calls to <code>SQLSetPos</code> with the <code>LockType</code> argument set to <code>SQL_LOCK_EXCLUSIVE</code> or <code>SQL_LOCK_UNLOCK</code> is not a part of the conformance levels but is an optional feature.)</td>
</tr>
<tr>
<td>207</td>
<td>Enable asynchronous execution of ODBC functions for specified individual statements.</td>
</tr>
<tr>
<td>208</td>
<td>Obtain the <code>SQL_ROWVER</code> row-identifying column of tables, by calling <code>SQLSpecialColumns</code>. (For more information, see the support for <code>SQLSpecialColumns</code> with the <code>IdentifierType</code> argument set to <code>SQL.BEST_ROWID</code> as feature 20 in Core Interface Conformance.)</td>
</tr>
<tr>
<td>209</td>
<td>Set the <code>SQL_ATTR_CONCURRENCY</code> statement attribute to at least one value other than <code>SQL_CONCUR_READ_ONLY</code>.</td>
</tr>
<tr>
<td>210</td>
<td>The ability to time out login request and SQL queries (<code>SQL_ATTR_LOGIN_TIMEOUT</code> and <code>SQL_ATTR_QUERY_TIMEOUT</code>).</td>
</tr>
<tr>
<td>211</td>
<td>The ability to change the default isolation level; the ability to execute transactions with the &quot;serializable&quot; level of isolation.</td>
</tr>
</tbody>
</table>

Function Conformance
The following table indicates the conformance level of each ODBC function, where this is well defined.

<table>
<thead>
<tr>
<th>Function</th>
<th>Conformance level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocHandle</td>
<td>Core</td>
</tr>
<tr>
<td>SQLBindCol</td>
<td>Core</td>
</tr>
<tr>
<td>SQLBindParameter</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQLBrowseConnect</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLBulkOperations</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLCancel</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQLCloseCursor</td>
<td>Core</td>
</tr>
<tr>
<td>SQLColAttribute</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQLColumnPrivileges</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLColumns</td>
<td>Core</td>
</tr>
<tr>
<td>SQLConnect</td>
<td>Core</td>
</tr>
<tr>
<td>SQLCopyDesc</td>
<td>Core</td>
</tr>
<tr>
<td>SQLDataSources</td>
<td>Core</td>
</tr>
<tr>
<td>SQLErrorCol</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQLDescribeCol</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLDisconnect</td>
<td>Core</td>
</tr>
<tr>
<td>SQLDriverConnect</td>
<td>Core</td>
</tr>
<tr>
<td>SQLDrivers</td>
<td>Core</td>
</tr>
<tr>
<td>SQLEndTran</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQLExecDirect</td>
<td>Core</td>
</tr>
<tr>
<td>SQLExecute</td>
<td>Core</td>
</tr>
<tr>
<td>SQLFetch</td>
<td>Core</td>
</tr>
<tr>
<td>SQLFetchScroll</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQLForeignKeys</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLFreeHandle</td>
<td>Core</td>
</tr>
<tr>
<td>SQLFreeStmt</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetConnectAttr</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetCursorName</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetData</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetDescField</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetDescRec</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetDiagField</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetDiagRec</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetEnvAttr</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetFunctions</td>
<td>Core</td>
</tr>
<tr>
<td>Function</td>
<td>Conformance level</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>SQLGetInfo</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetStmtAttr</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetTypeInfo</td>
<td>Core</td>
</tr>
<tr>
<td>SQLMoreResults</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLNativeSql</td>
<td>Core</td>
</tr>
<tr>
<td>SQLNumParams</td>
<td>Core</td>
</tr>
<tr>
<td>SQLNumResultCols</td>
<td>Core</td>
</tr>
<tr>
<td>SQLParamData</td>
<td>Core</td>
</tr>
<tr>
<td>SQLPrepare</td>
<td>Core</td>
</tr>
<tr>
<td>SQLPrimaryKeys</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLProcedureColumns</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLProcedures</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLPutData</td>
<td>Core</td>
</tr>
<tr>
<td>SQLRowCount</td>
<td>Core</td>
</tr>
<tr>
<td>SQLSetConnectAttr</td>
<td>Core[2]</td>
</tr>
<tr>
<td>SQLSetCursorName</td>
<td>Core</td>
</tr>
<tr>
<td>SQLSetDescField</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQLSetDescRec</td>
<td>Core</td>
</tr>
<tr>
<td>SQLSetEnvAttr</td>
<td>Core[2]</td>
</tr>
<tr>
<td>SQLSetPos</td>
<td>Level 1[1]</td>
</tr>
<tr>
<td>SQLSetStmtAttr</td>
<td>Core[2]</td>
</tr>
<tr>
<td>SQLSpecialColumns</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQLStatistics</td>
<td>Core</td>
</tr>
<tr>
<td>SQLTablePrivileges</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLTables</td>
<td>Core</td>
</tr>
</tbody>
</table>

[1] Significant features of this function are available only at higher conformance levels.

[2] Setting certain attributes to nondefault values depends on the conformance level. For more information, see the next section, Attribute Conformance.

© 2016 Microsoft

**Attribute Conformance**

The following table indicates the conformance level of each ODBC environment attribute, where this is well defined.
The following table indicates the conformance level of each ODBC connection attribute, where this is well defined.

<table>
<thead>
<tr>
<th>Function</th>
<th>Conformance level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_ACCESS_MODE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_ENABLE</td>
<td>Level 1/Level 2[1]</td>
</tr>
<tr>
<td>SQL_ATTR_AUTO_IPD</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_ATTR_AUTOCOMMIT</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQL_ATTR_CONNECTION_DEAD</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQL_ATTR_CONNECTION_TIMEOUT</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_ATTR_CURRENT_CATALOG</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_ATTR_LOGIN_TIMEOUT</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_ATTR_ODBC_CURSORS</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_PACKET_SIZE</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_ATTR_QUIET_MODE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_TRACE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_TRACEFILE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_TRANSLATE_LIB</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_TRANSLATE_OPTION</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_TXN_ISOLATION</td>
<td>Level 1/Level 2[2]</td>
</tr>
</tbody>
</table>

[1] Applications that support connection-level asynchrony (required for Level 1) must support setting this attribute to SQL_TRUE by calling SQLSetConnectAttr; the attribute need not be settable to a value other than its default value through SQLSetStmtAttr. Applications that support statement-level asynchrony (required for Level 2) must support setting this attribute to SQL_TRUE using either function.

[2] For Level 1 interface conformance, the driver must support one value in addition to the driver-defined default value (available by calling SQLGetInfo with the SQL_DEFAULT_TXN_ISOLATION option). For Level 2 interface conformance, the driver must also support SQL_TXN_SERIALIZABLE.

The following table indicates the conformance level of each ODBC statement attribute, where this is well defined.

<table>
<thead>
<tr>
<th>Function</th>
<th>Conformance level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_APP_PARAM_DESC</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_APP_ROW_DESC</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_ENABLE</td>
<td>Level 1/Level 2[1]</td>
</tr>
<tr>
<td>SQL_ATTR_CONCURRENCY</td>
<td>Level 1/Level 2[2]</td>
</tr>
<tr>
<td>SQL_ATTR_CURSOR_SCROLLABLE</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQL_ATTR_CURSOR_SENSITIVITY</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_ATTR_CURSOR_TYPE</td>
<td>Core/Level 2[3]</td>
</tr>
<tr>
<td>SQL_ATTR_ENABLE_AUTO_IPD</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_ATTR_FETCH_BOOKMARK_PTR</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_ATTR_IMP_PARAM_DESC</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_IMP_ROW_DESC</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_ATTR_KEYSET_SIZE</td>
<td>Level 2</td>
</tr>
</tbody>
</table>
### SQL_ATTR_MAX_LENGTH
- **Level 1**

### SQL_ATTR_MAX_ROWS
- **Level 1**

### SQL_ATTR_METADATA_ID
- **Core**

### SQL_ATTR_NOSCAN
- **Core**

### SQL_ATTR_PARAM_BIND_OFFSET_PTR
- **Core**

### SQL_ATTR_PARAM_BIND_TYPE
- **Core**

### SQL_ATTR_PARAM_OPERATION_PTR
- **Core**

### SQL_ATTR_PARAM_STATUS_PTR
- **Core**

### SQL_ATTR_PARAMS_PROCESSED_PTR
- **Core**

### SQL_ATTR_PARAMSET_SIZE
- **Core**

### SQL_ATTR_QUERY_TIMEOUT
- **Level 2**

### SQL_ATTR_RETRIEVE_DATA
- **Level 1**

### SQL_ATTR_ROW_ARRAY_SIZE
- **Core**

### SQL_ATTR_ROW_BIND_OFFSET_PTR
- **Core**

### SQL_ATTR_ROW_BIND_TYPE
- **Core**

### SQL_ATTR_ROW_NUMBER
- **Level 1**

### SQL_ATTR_ROW_OPERATION_PTR
- **Level 1**

### SQL_ATTR_ROW_STATUS_PTR
- **Core**

### SQL_ATTR_ROWS_FETCHED_PTR
- **Core**

### SQL_ATTR_SIMULATE_CURSOR
- **Level 2**

### SQL_ATTR_USE_BOOKMARKS
- **Level 2**

---

[1] Applications that support connection-level asynchrony (required for Level 1) must support setting this attribute to SQL_TRUE by calling **SQLSetConnectAttr**; the attribute need not be settable to a value other than its default value through **SQLSetStmtAttr**. Applications that support statement-level asynchrony (required for Level 2) must support setting this attribute to SQL_TRUE using either function.

[2] For Level 2 interface conformance, the driver must support SQL_CONCUR_READ_ONLY and at least one other value.

[3] For Level 1 interface conformance, the driver must support SQL_CURSOR_FORWARD_ONLY and at least one other value. For Level 2 interface conformance, the driver must support all values defined in this document.

---

© 2016 Microsoft

## Descriptor Field Conformance

The following table indicates the conformance level of each ODBC descriptor header field, where this is well defined.

<table>
<thead>
<tr>
<th>Function</th>
<th>Conformance level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_ALLOC_TYPE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_ARRAY_SIZE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_ARRAY_STATUS_PTR</td>
<td>Core (for APD, IPR, and IRD); Level 1 (for ARD)</td>
</tr>
<tr>
<td>SQL_DESC_BIND_OFFSET_PTR</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_BIND_TYPE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_COUNT</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_ROWS_PROCESSED_PTR</td>
<td>Core</td>
</tr>
</tbody>
</table>
The following table indicates the conformance level of each ODBC descriptor record field, where this is well defined.

<table>
<thead>
<tr>
<th>Function</th>
<th>Conformance level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_AUTO_UNIQUE_VALUE</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_DESC_BASE_COLUMN_NAME</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_BASE_TABLE_NAME</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQL_DESC_CASE_SENSITIVE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_CATALOG_NAME</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_DESC_CONCISE_TYPE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_DATA_PTR</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_DATETIME_INTERVAL_CODE</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQL_DESC_DATETIME_INTERVAL_PRECISION</td>
<td>Core[1]</td>
</tr>
<tr>
<td>SQL_DESC_DISPLAY_SIZE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_FIXED_PREC_SCALE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_INDICATOR_PTR</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_LABEL</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQL_DESC_LENGTH</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_LITERAL_PREFIX</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_LITERAL_SUFFIX</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_LOCAL_TYPE_NAME</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_NAME</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_NULLABLE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_OCTET_LENGTH</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_OCTET_LENGTH_PTR</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_PARAMETER_TYPE</td>
<td>Core/Level 2[2]</td>
</tr>
<tr>
<td>SQL_DESC_PRECISION</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_ROWVER</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQL_DESC_SCALE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_SCHEMA_NAME</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQL_DESC_SEARCHABLE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_TABLE_NAME</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQL_DESC_TYPE</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_TYPE_NAME</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_UNNAMED</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_UNSIGNED</td>
<td>Core</td>
</tr>
<tr>
<td>SQL_DESC_UPDATABLE</td>
<td>Core</td>
</tr>
</tbody>
</table>

[1] Support for these record fields is required only if the driver supports the applicable data types.
For Core-level conformance, the driver must support SQL_PARAM_INPUT. For Level 2 interface conformance, the driver must also support SQL_PARAM_INPUT_OUTPUT and SQL_PARAM_OUTPUT.

SQL Conformance Levels

The level of SQL-92 grammar supported by a driver is indicated by the value returned by a call to SQLGetInfo with the SQL_SQL_CONFORMANCE information type. This indicates whether the driver conforms to the Entry, FIPS Transitional, Intermediate, or Full levels defined in SQL-92.

All ODBC drivers must support the minimum SQL grammar described in SQL Minimum Grammar in Appendix C: SQL Grammar. This grammar is a subset of the Entry level of SQL-92. Drivers may support additional SQL and be conformant to the SQL-92 Entry, Intermediate, or Full level, or to the FIPS 127-2 Transitional level. Drivers that comply to a given level of SQL-92 or FIPS 127-2 can support additional features in any of the higher levels yet not be fully conformant to that level. To determine whether a feature is supported, an application should call SQLGetInfo with the appropriate information type. The conformance level of an SQL feature is described in the corresponding information type. (See the SQLGetInfo function description.)

Environment, Connection, and Statement Attributes

ODBC defines a number of attributes that are associated with environments, connections, or statements.

Environment attributes affect the entire environment, such as whether connection pooling is enabled. Environment attributes are set with SQLSetEnvAttr and retrieved with SQLGetEnvAttr.

Connection attributes affect each connection individually, such as how long a driver should wait while attempting to connect to a data source before timing out. Connection attributes are set with SQLSetConnectAttr and retrieved with SQLGetConnectAttr. For more information about connection attributes, see Connection Attributes.

Statement attributes affect each statement individually, such as whether a statement should be executed asynchronously. Statement attributes are set with SQLSetStmtAttr and retrieved with SQLGetStmtAttr. A few statement attributes are read-only attributes and cannot be set. For example, the SQL_ATTR_ROW_NUMBER statement attribute, which is used to retrieve the number of the current row in the cursor, is read-only. For more information about statement attributes, see Statement Attributes.

In addition to attributes defined by ODBC, a driver can define its own connection and statement attributes. Driver-defined attributes must be registered with Open Group to ensure that two driver vendors do not assign the same integer value to different, proprietary attributes. For more information, see Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes.

For a complete list of attributes, see SQLSetEnvAttr, SQLSetConnectAttr, and SQLSetStmtAttr. Most attributes are also described in the description of the ODBC function that they affect.

Tables and Views

In ODBC functions, tables and views are interchangeable. The term table is used both for tables and for views, except where the term view is used explicitly.

Basic ODBC Application Steps

This section describes the general flow of ODBC applications. It is unlikely that any application calls all of these functions in exactly this order. However, most applications use some variation of these steps. The basic application steps are shown in the following illustration.
Step 1: Connect to the Data Source

The first step in any application is to connect to the data source. This phase, including the functions it requires, is shown in the following illustration.

The first step in connecting to the data source is to load the Driver Manager and allocate the environment handle with `SQLAllocHandle`. For more information, see Allocating the Environment Handle.
The application then registers the version of ODBC to which it conforms by calling `SQLSetEnvAttr` with the `SQL_ATTR_APP_ODBC_VER` environment attribute. For more information, see Declaring the Application’s ODBC Version and Backward Compatibility and Standards Compliance.

Next, the application allocates a connection handle with `SQLAllocHandle` and connects to the data source with `SQLConnect`, `SQLDriverConnect`, or `SQLBrowseConnect`. For more information, see Allocating a Connection Handle and Establishing a Connection.

The application then sets any connection attributes, such as whether to manually commit transactions. For more information, see Connection Attributes.

### Step 2: Initialize the Application

The second step is to initialize the application, as shown in the following illustration. Exactly what is done here varies with the application.

At this point, it is common to use `SQLGetInfo` to discover the capabilities of the driver. For more information, see Considering Database Features to Use.

All applications need to allocate a statement handle with `SQLAllocHandle`, and many applications set statement attributes, such as the cursor type, with `SQLSetStmtAttr`. For more information, see Allocating a Statement Handle and Statement Attributes.

### Step 3: Build and Execute an SQL Statement

The third step is to build and execute an SQL statement, as shown in the following illustration. The methods used to perform this step are likely to vary tremendously. The application might prompt the user to enter an SQL statement, build an SQL statement based on user input, or use a hard-coded SQL statement. For more information, see Constructing SQL Statements.

If the SQL statement contains parameters, the application binds them to application variables by calling `SQLBindParameter` for each parameter. For more information, see Statement Parameters.

After the SQL statement is built and any parameters are bound, the statement is executed with `SQLExecDirect`. If the statement will be executed multiple times, it can be prepared with `SQLPrepare` and executed with `SQLExecute`. For more information, see Executing a Statement.

The application might also forgo executing an SQL statement altogether and instead call a function to return a result set containing catalog information, such as the available columns or tables. For more information, see Uses of Catalog Data.

The application’s next action depends on the type of SQL statement executed.

<table>
<thead>
<tr>
<th>Type of SQL statement</th>
<th>Proceed to</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT</strong> or catalog function</td>
<td>Step 4a: Fetch the Results</td>
</tr>
<tr>
<td><strong>UPDATE, DELETE, or INSERT</strong></td>
<td>Step 4b: Fetch the Row Count</td>
</tr>
<tr>
<td>All other SQL statements</td>
<td>Step 3: Build and Execute an SQL Statement (this topic) or Step 5: Commit the Transaction</td>
</tr>
</tbody>
</table>

### Step 4a: Fetch the Results

The next step is to fetch the results, as shown in the following illustration.
If the statement executed in “Step 3: Build and Execute an SQL Statement” was a SELECT statement or a catalog function, the application first calls SQLNumResultCols to determine the number of columns in the result set. This step is not necessary if the application already knows the number of result set columns, such as when the SQL statement is hard-coded in a vertical or custom application.

Next, the application retrieves the name, data type, precision, and scale of each result set column with SQLDescribeCol. Again, this is not necessary for applications such as vertical and custom applications that already know this information. The application passes this information to SQLBindCol, which binds an application variable to a column in the result set.

The application now calls SQLEndTran to commit or roll back the transaction. The application performs this step only if it set the transaction commit mode to manual-commit; if the transaction commit mode is auto-commit, which is the default, the transaction is automatically committed when the statement is executed. For more information, see Transactions.

To execute a statement in a new transaction, the application returns to step 3. To disconnect from the data source, the application proceeds to step 6.

The application now returns to “Step 3: Build and Execute an SQL Statement” to execute another statement in the same transaction; or proceeds to “Step 5: Commit the Transaction” to commit or roll back the transaction.

For a complete description of retrieving results, see Retrieving Results (Basic) and Retrieving Results (Advanced).

The application now returns to Step 3 to execute another statement in the same transaction or proceeds to step 5 to commit or roll back the transaction.

© 2016 Microsoft

Step 4b: Fetch the Row Count

The next step is to fetch the row count, as shown in the following illustration.

If the statement executed in Step 3 was an UPDATE, DELETE, or INSERT statement, the application retrieves the count of affected rows with SQLRowCount. For more information, see Determining the Number of Affected Rows.

The application now returns to step 3 to execute another statement in the same transaction or proceeds to step 5 to commit or roll back the transaction.

© 2016 Microsoft

Step 5: Commit the Transaction

The next step is to commit the transaction, as shown in the following illustration.

The fifth step is to call SQLEndTran to commit or roll back the transaction. The application performs this step only if it set the transaction commit mode to manual-commit; if the transaction commit mode is auto-commit, which is the default, the transaction is automatically committed when the statement is executed. For more information, see Transactions.

To execute a statement in a new transaction, the application returns to step 3. To disconnect from the data source, the application proceeds to step 6.

© 2016 Microsoft

Step 6: Disconnect from the Data Source

The final step is to disconnect from the data source, as shown in the following illustration. First, the application frees any statement handles by calling SQLFreeHandle. For more information, see Freeing a Statement Handle.

Next, the application disconnects from the data source with SQLDisconnect and frees the connection handle with SQLFreeHandle. For more information, see Disconnecting from a Data Source or Driver.
Finally, the application frees the environment handle with SQLFreeHandle and unloads the Driver Manager. For more information, see Allocating the Environment Handle.

Connecting to a Data Source or Driver

An application can be connected to any number of drivers and data sources. These can be a variety of drivers and data sources, the same driver and a variety of data sources, or even multiple connections to the same driver and data source.

This section contains the following topics.

- Allocating the Environment Handle
- Declaring the Application's ODBC Version
- Choosing a Data Source or Driver
- Allocating a Connection Handle
- Connection Attributes
- Establishing a Connection
- Driver Manager Connection Pooling
- Disconnecting from a Data Source or Driver
- The Driver Manager's Role in the Connection Process

Allocating the Environment Handle

The first task for any ODBC application is to load the Driver Manager; how this is done is operating-system dependent. For example, on a computer running Microsoft® Windows NT® Server/Windows 2000 Server, Windows NT Workstation/Windows 2000 Professional, or Microsoft Windows® 95/98, the application either links to the Driver Manager library or calls LoadLibrary to load the Driver Manager DLL.

The next task, which must be done before an application can call any other ODBC function, is to initialize the ODBC environment and allocate an environment handle, as follows:

1. The application declares a variable of type SQLHENV. It then calls SQLAllocHandle and passes the address of this variable and the SQL_HANDLE_ENV option. For example:

   ```c
   SQLHENV henv1;
   SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv1);
   ```

2. The Driver Manager allocates a structure in which to store information about the environment, and returns the environment handle in the variable.

The Driver Manager does not call SQLAllocHandle in the driver at this time because it does not know which driver to call. It delays calling SQLAllocHandle in the driver until the application calls a function to connect to a data source. For more information, see Driver Manager's Role in the Connection Process, later in this section.

When the application has finished using ODBC, it frees the environment handle with SQLFreeHandle. After freeing the environment, it is an application programming error to use the environment's handle in a call to an ODBC function; doing so has undefined but probably fatal consequences.

When SQLFreeHandle is called, the driver releases the structure used to store information about the environment. Note that SQLFreeHandle cannot be called for an environment handle until after all connection handles on that environment handle have been freed.

For more information about the environment handle, see Environment Handles.

Declaring the Application's ODBC Version

Before an application allocates a connection, it must set the SQL_ATTR_ODBC_VERSION environment attribute. This attribute states that the application follows the ODBC 2.x or ODBC 3.x specification when using the following items:
**SQLSTATEs.** Many SQLSTATE values are different in ODBC 2.x and ODBC 3.x.

**Date, Time, and Timestamp Type Identifiers.** The following table shows the type identifiers for date, time, and timestamp data in ODBC 2.x and ODBC 3.x.

<table>
<thead>
<tr>
<th>ODBC 2.x</th>
<th>ODBC 3.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DATE</td>
<td>SQL_TYPE_DATE</td>
</tr>
<tr>
<td>SQL_TIME</td>
<td>SQL_TYPE_TIME</td>
</tr>
<tr>
<td>SQL_TIMESTAMP</td>
<td>SQL_TYPE_TIMESTAMP</td>
</tr>
<tr>
<td>SQL_C_DATE</td>
<td>SQL_C_TYPE_DATE</td>
</tr>
<tr>
<td>SQL_C_TIME</td>
<td>SQL_C_TYPE_TIME</td>
</tr>
<tr>
<td>SQL_C_TIMESTAMP</td>
<td>SQL_C_TYPE_TIMESTAMP</td>
</tr>
</tbody>
</table>

**CatalogName Argument in SQTables.** In ODBC 2.x, the wildcard characters ("%" and "_") in the CatalogName argument are treated literally. In ODBC 3.x, they are treated as wildcard characters. Thus, an application that follows the ODBC 2.x specification cannot use these as wildcard characters and does not escape them when using them as literals. An application that follows the ODBC 3.x specification can use these as wildcard characters or escape them and use them as literals. For more information, see Arguments in Catalog Functions.

The ODBC 3.x Driver Manager and ODBC 3.x drivers check the version of the ODBC specification to which an application is written and respond accordingly. For example, if the application follows the ODBC 2.x specification and calls SQLExecute before calling SQLPrepare, the ODBC 3.x Driver Manager returns SQLSTATE S1010 (Function sequence error). If the application follows the ODBC 3.x specification, the Driver Manager returns SQLSTATE HY010 (Function sequence error). For more information, see Backward Compatibility and Standards Compliance.

### Important

Applications that follow the ODBC 3.x specification must use conditional code to avoid using functionality new to ODBC 3.x when working with ODBC 2.x drivers. ODBC 2.x drivers do not support functionality new to ODBC 3.x just because the application declares that it follows the ODBC 3.x specification. Furthermore, ODBC 3.x drivers do not cease to support functionality new to ODBC 3.x just because the application declares that it follows the ODBC 2.x specification.

© 2016 Microsoft

---

### Choosing a Data Source or Driver

The data source or driver used by an application is sometimes hard-coded in the application. For example, a custom application written by an MIS department to transfer data from one data source to another would contain the names of those data sources—the application simply would not work with any other data sources. Another example is a vertical application, such as one used for order entry. Such an application always uses the same data source, which has a predefined schema known by the application.

Other applications select the data source or driver at run time. Usually, these are generic applications that do ad hoc queries, such as a spreadsheet that uses ODBC to import data. Such applications usually list the available data sources or drivers and let users choose the ones they want to work with. Whether a generic application lists data sources, drivers, or both frequently depends on whether the application uses DBMS-based or file-based drivers.

DBMS-based drivers usually require a complex set of connection information, such as the network address, network protocol, database name, and so on. The purpose of a data source is to hide all of this information. Therefore, the data source paradigm lends itself to use with DBMS-based drivers. An application can display a list of data sources to the user in one of two ways. It can call SQDDriverConnect with the DSN (Data Source Name) keyword and no associated value; the Driver Manager will display a list of data source names. If the application wants control over the appearance of the list, it calls SQLDataSources to retrieve a list of available data sources and constructs its own dialog box. This function is implemented by the Driver Manager and can be called before any drivers are loaded. The application then calls a connection function and passes it the name of the chosen data source.

If a data source is not specified, the default data source indicated by the system information is used. (For more information, see Default Subkey.) If SQLConnect is called by using a ServerName argument that cannot be found, is a null pointer, or is "DEFAULT", the Driver Manager connects to the default data source. The default data source is also used if the connection string that is used in a call to SQDDriverConnect or SQLBrowseConnect contains the DSN keyword set to "DEFAULT" or if the specified data source is not found. Additionally, the default data source is used if the connection string that is used in a call to SQDDriverConnect does not contain the DSN keyword.

With file-based drivers, it is possible to use a file paradigm. For data stored on the local computer, users frequently know that their data is in a particular file, such as Employee.dbf. Instead of selecting an unknown data source, it is easier for such users to select the file they know. To implement this, the application first calls SQLDrivers. This function is implemented by the Driver Manager and can be called before any drivers are loaded. SQLDrivers returns a list of available drivers; it also returns values for the FileUsage and FileExtns keywords. The FileUsage keyword explains whether file-based drivers treat files as tables, as does Xbase, or as databases, as does Microsoft® Access. The FileExtns keyword lists the file name extensions the driver recognizes, such as .dbf for an Xbase driver. Using this information, the application constructs a dialog box through which the user chooses a file. Based on the extension of the chosen file, the application then connects to the driver by calling SQDDriverConnect with the DRIVER keyword to connect...
to a DBMS-based driver. Here are several common uses of the DRIVER keyword for DBMS-based drivers:

- **Not creating data sources.** For example, a custom application might use a particular driver and database. If the driver name and all information that is required to connect to the database is hard-coded in the application, users do not have to create a data source on their computer to run the application. All they must do is install the application and driver.

  A disadvantage of this method is that the application must be recompiled and redistributed if the connection information changes. If a data source name is hard-coded in the application instead of complete connection information, each user must change only the information in the data source.

- **Accessing a particular DBMS a single time.** For example, a spreadsheet that retrieves data by calling ODBC functions might contain the DRIVER keyword to identify a particular driver. Because the driver name is meaningful to any users who have that driver, the spreadsheet could be passed among those users. If the spreadsheet contained a data source name, each user would have to create the same data source to use the spreadsheet.

- **Browsing the system for all databases accessible to a particular driver.** For more information, see Connecting with SQLBrowseConnect, later in this section.

© 2016 Microsoft

## Allocating a Connection Handle ODBC

Before the application can connect to a data source or driver, it must allocate a connection handle, as follows:

1. The application declares a variable of type SQLHDBC. It then calls SQLAllocHandle and passes the address of this variable, the handle of the environment in which to allocate the connection, and the SQL_HANDLE_DBC option. For example:

   ```
   SQLHDBC hdbc1;
   SQLAllocHandle(SQL_HANDLE_DBC, henv1, &hdbc1);
   ```

2. The Driver Manager allocates a structure in which to store information about the statement and returns the connection handle in the variable.

   The Driver Manager does not call SQLAllocHandle in the driver at this time because it does not know which driver to call. It delays calling SQLAllocHandle in the driver until the application calls a function to connect to a data source. For more information, see Driver Manager’s Role in the Connection Process, later in this section.

   It is important to note that allocating a connection handle is not the same as loading a driver. The driver is not loaded until a connection function is called. Thus, after allocating a connection handle and before connecting to the driver or data source, the only functions the application can call with the connection handle are SQLSetConnectAttr, SQLGetConnectAttr, or SQLGetInfo with the SQL_ODBC_VER option. Calling other functions with the connection handle, such as SQLEndTran, returns SQLSTATE 08003 (Connection not open). For complete details, see Appendix B: ODBC State Transition Tables.

   For more information about connection states, see Connection Handles.

© 2016 Microsoft

## Connection Attributes

Connection attributes are characteristics of the connection. For example, because transactions occur at the connection level, the transaction isolation level is a connection attribute. Similarly, the login timeout, 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 retrieved with SQLGetConnectAttr. If SQLSetConnectAttr is called before the driver is loaded, the Driver Manager stores the attributes in its connection structure and sets them in the driver as part of the connection process. There is no requirement that an application set any connection attributes; all connection attributes have defaults, some of which are driver-specific.

A connection attribute can be set before or after connection, or either, depending on the attribute and the driver. The login timeout (SQL_ATTR_LOGIN_TIMEOUT) applies to the connection process and is effective only if set before connecting. The attributes that specify whether to use the ODBC cursor library (SQL_ATTR_ODBC_CURSORS) and the network packet size (SQL_ATTR_PACKET_SIZE) must be set before connecting, because the ODBC cursor library resides between the Driver Manager and the driver and therefore must be loaded before the driver.

The attributes to specify whether a data source is read-only or read-write (SQL_ATTR_ACCESS_MODE) and the current catalog (SQL_ATTR_CURRENT_CATALOG) can be set before or after connecting, depending on the driver. However, interoperable applications set them before connecting because some drivers do not support changing these after connecting.

Some connection attributes have a default before the connection is made, while others do not. Those that do are SQL_ATTR_ACCESS_MODE, SQL_ATTR_AUTOCOMMIT, SQL_ATTR_LOGIN_TIMEOUT, SQL_ATTR_ODBC_CURSORS, SQL_ATTR_TRACE, and SQL_ATTR_TRACEFILE.

The translation connection attributes (SQL_ATTR_TRANSLATE_DLL and SQL_ATTR_TRANSLATE_OPTION) must be set after connecting.

All other connection attributes can be set at any time. For more information, see the SQLSetConnectAttr function description. (Connection attributes cannot be set on the environment level by a call to SQLSetEnvAttr.)

© 2016 Microsoft
Establishing a Connection

After allocating environment and connection handles and setting any connection attributes, the application is ready to connect to the data source or driver. There are three different functions the application can use to do this: SQLConnect (Core interface conformance level), SQLDriverConnect (Core), and SQLBrowseConnect (Level 1). Each of the three is designed to be used in a different scenario. Before connecting, the application can determine which of these functions is supported with the ConnectFunctions keyword returned by SQLDrivers.

Note

Some drivers limit the number of active connections they support. An application calls SQLGetInfo with the SQL_MAX_DRIVER_CONNECTIONS option to determine how many active connections a particular driver supports.

This section contains the following topics.

- Default Data Source
- Connecting with SQLConnect
- Connection Strings
- Connecting with SQLDriverConnect
- Connecting with SQLBrowseConnect

Default Data Source

The driver may select a data source, called the default data source, in certain cases where the application does not explicitly specify one:

- In a call to SQLConnect where the ServerName argument is a zero-length string, a null pointer, or DEFAULT.
- In a call to SQLDriverConnect where InConnectionString either specifies DSN=DEFAULT or specifies with the DSN keyword a data source that is not contained in the system information.

It is driver-defined how the default data source is specified. This may involve administrative action and may depend on the user.

Connecting with SQLConnect

SQLConnect is the simplest connection function. It requires a data source name and accepts an optional user ID and password. It works well for applications that hard-code a data source name and do not require a user ID or password. It also works well for applications that want to control their own "look and feel" or that have no user interface. Such applications can build a list of data sources using SQLDataSources, prompt the user for data source, user ID, and password, and then call SQLConnect.

The following example connects to the Northwind database, using a DSN called Northwind, and retrieves all of the first and last name fields from all of the records in the Employees table.

```c++
// Connecting_with_SQLConnect.cpp
// compile with: user32.lib odbc32.lib
#include <windows.h>
#include <sqlext.h>
#include <mbstring.h>
#include <stdio.h>
#define MAX_DATA 100
#define MYSQLSUCCESS(rc) ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO))

class direxec {  
  RETCODE rc; // ODBC return code  
  HENV henv; // Environment  
  HDBC hdc; // Connection handle  
  HSTMT hstmt; // Statement handle  

  unsigned char szData[MAX_DATA]; // Returned data storage  
  SDWORD cbData; // Output length of data
```
unsigned char chr_ds_name[SQL_MAX_DSN_LENGTH]; // Data source name

public:

direxec(); // Constructor
void sqlconn(); // Allocate env, stat, and conn
void sqlexecute(unsigned char *); // Execute SQL statement
void sqldisconnect(); // Free pointers to env, stat, conn, and disconnect
void error_out(); // Displays errors
};

// Constructor initializes the string chr_ds_name with the data source name.
// "Northwind" is an ODBC data source (odbcad32.exe) name whose default is the Northwind database

direxec::direxec()
{
_mbscpy_s(chr_ds_name, SQL_MAX_DSN_LENGTH, (const unsigned char*)"Northwind");
}

// Allocate environment handle and connection handle, connect to data source, and allocate statement handle.
void direxec::sqlconn()
{
SQLAllocEnv(&henv);
SQLAllocConnect(henv, &hdbc);
rc = SQLConnect(hdbc, chr_ds_name, SQL_NTS, NULL, 0, NULL, 0);

// Deallocate handles, display error message, and exit.
if (!MYSQLSUCCESS(rc)) {
SQLFreeConnect(henv);
SQLFreeEnv(henv);
SQLFreeConnect(hdbc);
if (hstmt)
error_out();
exit(-1);
}

rc = SQLAlloc Stmt(hdbc, &hstmt);
}

// Execute SQL command with SQLExecDirect() ODBC API.
void direxec::sqlexecute(unsigned char * cmdstr)
{
rc = SQLExecDirect(hstmt, cmdstr, SQL_NTS);
if (!MYSQLSUCCESS(rc)) {
// Error
error_out();
// Deallocate handles and disconnect.
SQLFreeStmt(hstmt, SQL_DROP);
SQLDisconnect(hdbc);
SQLFreeConnect(hdbc);
SQLFreeEnv(henv);
exit(-1);
}

for (
rc = SQLFetch(hstmt) ; rc == SQL_SUCCESS ; rc = SQLFetch(hstmt) )
{
SQLGetData(hstmt, 1, SQL_C_CHAR, szData, sizeof(szData), &cbData);
// In this example, the data is sent to the console; SQLBindCol() could be called to bind
// individual rows of data and assign for a rowset.
printf("%s\n", (const char*)szData);
}
}

// Free the statement handle, disconnect, free the connection handle, and free the environment handle.
void direxec::sqldisconnect()
{
SQLFreeStmt(hstmt, SQL_DROP);
SQLDisconnect(hdbc);
SQLFreeConnect(hdbc);
SQLFreeEnv(henv);
}

// Display error message in a message box that has an OK button.
void direxec::error_out()
{
unsigned char szSQLSTATE[10];
SDWORD nErr;
unsigned char msg[SQL_MAX_MESSAGE_LENGTH + 1];
SWORD cbmsg;

while (SQLGetErrm(0, 0, hstmt, szSQLSTATE, &nErr, msg, sizeof(msg), &cbmsg) == SQL_SUCCESS) {
printf_s( "Error: %s\n", szSQLSTATE );
}

MessageBox(NULL, (const char*)szData, "ODBC Error", MB_OK);
}

int main()
{
direxec x; // Declare an instance of the direxec object.
x.sqlconn(); // Allocate handles, and connect.
x.sqlexecute(UCHAR FAR *)"SELECT FirstName, LastName FROM employees"); // Execute SQL command
x.sqldisconnect(); // Free handles and disconnect
Connection Strings

A connection string contains information used for establishing a connection. A complete connection string contains all the information needed to establish a connection. The connection string is a series of keyword/value pairs separated by semicolons. (For the complete syntax of a connection string, see the SQLDriverConnect function description.) The connection string is used by:

- SQLDriverConnect, which completes the connection string by interaction with the user.
- SQLBrowseConnect, which completes the connection string iteratively with the data source.

SQLConnect does not use a connection string; using SQLConnect is analogous to connecting using a connection string with exactly three keyword/value pairs (for data source name and, optionally, user ID and password).

Connecting with SQLDriverConnect

SQLDriverConnect is used to connect to a data source using a connection string. SQLDriverConnect is used instead of SQLConnect for the following reasons:

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

This section contains the following topics.

- Driver-Specific Connection Information
- Prompting the User for Connection Information
- Connecting Using File Data Sources
- Connecting Directly to Drivers

Driver-Specific Connection Information

SQLConnect assumes that a data source name, user ID, and password are sufficient to connect to a data source and that all other connection information can be stored on the system. This is frequently not the case. For example, a driver might need one user ID and password to log on to a server and a different user ID and password to log on to a DBMS. Because SQLConnect accepts a single user ID and password, this means that the other user ID and password must be stored with the data source information on the system if SQLConnect is to be used. This is a potential breach of security and should be avoided unless the password is encrypted.

SQLDriverConnect allows the driver to define an arbitrary amount of connection information in the keyword-value pairs of the connection string. For example, suppose a driver requires a data source name, a user ID and password for the server, and a user ID and password for the DBMS. A custom program that always uses the XYZ Corp data source might prompt the user for IDs and passwords and build the following set of keyword-value pairs, or connection string, to pass to SQLDriverConnect:

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you are connecting to a data source provider that supports Windows authentication, you should specify Trusted_Connection=yes instead of user ID and password information in the connection string.</td>
</tr>
</tbody>
</table>

```
DSN={MyDataSourceName};UID={MyUserID};PWD={MyServerPassword};UIDDBMS={MyDBMSUserID};PWDDBMS={MyDBMSUserPassword};
```

The DSN (Data Source Name) keyword names the data source, the UID and PWD keywords specify the user ID and password for the server, and the UIDDBMS and PWDDBMS keywords specify the user ID and password for the DBMS. Notice that the final semicolon is optional. SQLDriverConnect parses this string; uses the XYZ Corp data source name to retrieve additional connection information from the system, such as the server address; and logs on to the server and DBMS using the specified user IDs and passwords.

Keyword-value pairs in SQLDriverConnect must follow certain syntax rules. The keywords and their values should not contain the [{}\|;\*!=@] characters. The value of the DSN keyword cannot consist only of blanks and should not contain leading blanks. Because of the registry grammar, keywords and data
source names cannot contain the backslash (\) character. Spaces are not allowed around the equal sign in the keyword-value pair.

The FILEDSN keyword can be used in a call to SQLDriverConnect to specify the name of a file that contains data source information (see Connecting Using File Data Sources, later in this section). The SAVEFILE keyword can be used to specify the name of a .dsn file in which the keyword-value pairs of a successful connection made by the call to SQLDriverConnect will be saved. For more information about file data sources, see the SQLDriverConnect function description.

Prompting the User for Connection Information

If the application uses SQLConnect and needs to prompt the user for any connection information, such as a user name and password, it must do so itself. While this allows the application to control its “look and feel,” it might force the application to contain driver-specific code. This occurs when the application needs to prompt the user for driver-specific connection information. This presents an impossible situation for generic applications, which are designed to work with any and all drivers, including drivers that do not exist when the application is written.

SQLDriverConnect can prompt the user for connection information. For example, the custom program mentioned earlier could pass the following connection string to SQLDriverConnect:

```
DSN=XYZ_Corp;
```

The driver might then display a dialog box that prompts for user IDs and passwords, similar to the following illustration.

That the driver can prompt for connection information is particularly useful to generic and vertical applications. These applications should not contain driver-specific information, and having the driver prompt for the information it needs keeps that information out of the application. This is shown by the previous two examples. When the application passed only the data source name to the driver, the application did not contain any driver-specific information and was therefore not tied to a particular driver. When the application passed a complete connection string to the driver, it was tied to the driver that could interpret that string.

A generic application might take this one step further and not even specify a data source. When SQLDriverConnect receives an empty connection string, the Driver Manager displays the following dialog box.

After the user selects a data source, the Driver Manager constructs a connection string specifying that data source and passes it to the driver. The driver can then prompt the user for any additional information it needs.

The conditions under which the driver prompts the user are controlled by the DriverCompletion flag; there are options to always prompt, prompt if necessary, or never prompt. For a complete description of this flag, see the SQLDriverConnect function description.

Connecting Using File Data Sources

The connection information for a file data source is stored in a .dsn file. As a result, the connection string can be used repeatedly by a single user or shared among several users if they have the appropriate driver installed. The file contains a driver name (or another data source name in the case of an unshareable file data source) and optionally, a connection string that can be used by SQLDriverConnect. The Driver Manager builds the connection string for the call to SQLDriverConnect from the keywords in the .dsn file.
A file data source allows an application to specify connection options without having to build a connection string for use with SQLDriverConnect. The file data source usually is created by specifying the SAVEFILE keyword, which causes the Driver Manager to save the output connection string created by a call to SQLDriverConnect to the .dsn file. That connection string can be used repeatedly by calling SQLDriverConnect with the FILEDSN keyword. This streamlines the connection process and provides a persistent source of the connection string.

File data sources also can be created by calling SQLCreateDataSource in the installer DLL. Information can be written into the .dsn file by calling SQLWriteFileDSN, and read from the .dsn file by calling SQLReadFileDSN; both of these functions are also in the installer DLL. For information about the installer DLL, see Configuring Data Sources.

The keywords used for connection information are in the [ODBC] section of a .dsn file. The minimum information that a shareable .dsn file would have in the [ODBC] section is the DRIVER keyword:

```
DRIVER = SQL Server
```

The shareable .dsn file usually contains a connection string, as follows:

```
DRIVER = SQL Server
UID = Larry
DATABASE = MyDB
```

When the file data source is unshareable, the .dsn file contains only a DSN keyword. When the Driver Manager is sent the information in an unshareable file data source, it connects as necessary to the data source indicated by the DSN keyword. An unshareable .dsn file would contain the following keyword:

```
DSN = MyDataSource
```

The connection string used for a file data source is the union of the keywords specified in the .dsn file and the keywords specified in the connection string in the call to SQLDriverConnect. If any of the keywords in the .dsn file conflict with keywords in the connection string, the Driver Manager decides which keyword value should be used. For more information, see SQLDriverConnect.

See Also

http://support.microsoft.com/kb/165866

© 2016 Microsoft

Connecting Directly to Drivers

As was discussed in Choosing a Data Source or Driver, earlier in this section, some applications do not want to use a data source at all. Instead, they want to connect directly to a driver. SQLDriverConnect provides a way for the application to connect directly to a driver without specifying a data source. Conceptually, a temporary data source is created at run time.

To connect directly to a driver, the application specifies the DRIVER keyword in the connection string instead of the DSN keyword. The value of the DRIVER keyword is the description of the driver as returned by SQLDrivers. For example, suppose a driver has the description Paradox Driver and requires the name of a directory containing the data files. To connect to this driver, the application might use either of the following connection strings:

```
DRIVER={Paradox Driver};Directory=C:\PARADOX;
DRIVER={Paradox Driver};
```

With the first string, the driver would not need any additional information. With the second string, the driver would need to prompt for the name of the directory containing the data files.

© 2016 Microsoft

Connecting with SQLBrowseConnect

SQLBrowseConnect, like SQLDriverConnect, uses a connection string. However, by using SQLBrowseConnect, an application can construct a complete connection string 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 "look and feel."
- 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.
The application calls **SQLBrowseConnect** and passes a connection string, known as the browse request connection string, that specifies a driver or data source. The driver returns a connection string, known as the browse result connection string, that contains keywords, possible values (if the keyword accepts a discrete set of values), and user-friendly names. The application builds a dialog box with the user-friendly names and prompts the user for values. It then builds a new browse request connection string from these values and returns this to the driver with another call to **SQLBrowseConnect**.

Because connection strings are passed back and forth, the driver can provide several levels of browsing by returning a new connection string when the application returns the old one. For example, the first time an application calls **SQLBrowseConnect**, the driver might return keywords to prompt the user for a server name. When the application returns the server name, the driver might return keywords to prompt the user for a database. The browsing process would be complete after the application returned the database name.

Each time **SQLBrowseConnect** returns a new browse result connection string, it returns SQL_NEED_DATA as its return code. This tells the application that the connection process is not complete. Until **SQLBrowseConnect** returns SQL_SUCCESS, the connection is in a Need Data state and cannot be used for other purposes, such as to set a connection attribute. The application can terminate the connection browsing process by calling **SQLDisconnect**.

This section contains the following topic:

- **SQL Server Browsing Example**

© 2016 Microsoft

## SQL Server Browsing Example

The following example shows how **SQLBrowseConnect** might be used to browse the connections available with a driver for SQL Server. First, the application requests a connection handle:

```sql
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
```

Next, the application calls **SQLBrowseConnect** and specifies the SQL Server driver, using the driver description returned by **SQLDrivers**:

```sql
SQLBrowseConnect(hdbc, "DRIVER={SQL Server};", SQL_NTS, BrowseResult, sizeof(BrowseResult), &BrowseResultLen);
```

Because this is the first call to **SQLBrowseConnect**, the Driver Manager loads the SQL Server driver and calls the driver's **SQLBrowseConnect** function with the same arguments it received from the application.

**Note**

If you are connecting to a data source provider that supports Windows authentication, you should specify **Trusted_Connection=yes** instead of user ID and password information in the connection string.

The driver determines that this is the first call to **SQLBrowseConnect** and returns the second level of connection attributes: server, user name, password, application name, and workstation ID. For the server attribute, it returns a list of valid server names. The return code from **SQLBrowseConnect** is SQL_NEED_DATA. Here is the browse result string:

```
"SERVER:Server={red,blue,green,yellow};UID:Login ID=?;PWD:Password=?;
 *APP:AppName=?;*WSID:WorkStation ID=?;"
```

Each keyword in the browse result string is followed by a colon and one or more words before the equal sign. These words are the user-friendly name that an application can use to build a dialog box. The **APP** and **WSID** keywords are prefixed by an asterisk, which means they are optional. The **SERVER**, **UID**, and **PWD** keywords are not prefixed by an asterisk; values must be supplied for them in the next browse request string. The value for the **SERVER** keyword may be one of the servers returned by **SQLBrowseConnect** or a user-supplied name.

The application calls **SQLBrowseConnect** again, specifying the green server and omitting the **APP** and **WSID** keywords and the user-friendly names after each keyword:

```sql
SQLBrowseConnect(hdbc, "SERVER=green;UID=Smith;PWD=Sesame;", SQL_NTS, BrowseResult, sizeof(BrowseResult), &BrowseResultLen);
```

The driver attempts to connect to the green server. If there are any nonfatal errors, such as a missing keyword-value pair, **SQLBrowseConnect** returns SQL_NEED_DATA and remains in the same state as it was prior to the error. The application can call **SQLGetDiagField** or **SQLGetDiagRec** to determine the error. If the connection is successful, the driver returns SQL_NEED_DATA and returns the browse result string:

```
"*DATABASE:Database={master,model,pubs,tempdb};
 *LANGUAGE:Language={us_english,Franais};"
```
Because the attributes in this string are optional, the application can omit them. However, the application must call `SQLBrowseConnect` again. If the application chooses to omit the database name and language, it specifies an empty browse request string. In this example, the application chooses the `pubs` database and calls `SQLBrowseConnect` a final time, omitting the `LANGUAGE` keyword and the asterisk before the `DATABASE` keyword:

```c
SQLBrowseConnect(hdbc, "DATABASE=pubs;", SQL_NTS, BrowseResult,
    sizeof(BrowseResult), &BrowseResultLen);
```

Because the `DATABASE` attribute is the final connection attribute required by the driver, the browsing process is complete, the application is connected to the data source, and `SQLBrowseConnect` returns `SQL_SUCCESS`. `SQLBrowseConnect` also returns the complete connection string as the browse result string:

```c
"DSN=MySQLServer;SERVER=green;UID=Smith;PWD=Sesame;DATABASE=pubs;"
```

The final connection string returned by the driver does not contain the user-friendly names after each keyword, nor does it contain optional keywords not specified by the application. The application can use this string with `SQLDriverConnect` to reconnect to the data source on the current connection handle (after disconnecting) or to connect to the data source on a different connection handle. For example:

```c
SQLDriverConnect(hdbc, hwnd, BrowseResult, SQL_NTS, ConnStrOut,
    sizeof(ConnStrOut), &ConnStrOutLen, SQL_DRIVER_NOPROMPT);
```

---

**Driver Manager Connection Pooling**

Connection pooling enables an application to use a connection from a pool of connections that do not need to be re-established for each use. Once a connection has been created and placed in a pool, an application can reuse that connection without performing the complete connection process.

Using a pooled connection can result in significant performance gains, because applications can save the overhead involved in making a connection. This can be particularly significant for middle-tier applications that connect over a network or for applications that repeatedly connect and disconnect, such as Internet applications.

In addition to performance gains, the connection pooling architecture enables an environment and its associated connections to be used by multiple components in a single process. This means that stand-alone components in the same process can interact with each other without being aware of each other. A connection in a connection pool can be used repeatedly by multiple components.

**Note**

Connection pooling can be used by an ODBC application exhibiting ODBC 2.x behavior, as long as the application can call `SQLSetEnvAttr`. When using connection pooling, the application must not execute SQL statements that change the database or the context of the database, such as changing the `<database name>`, which changes the catalog used by a data source.

An ODBC driver must be fully thread-safe, and connections must not have thread affinity to support connection pooling. This means the driver is able to handle a call on any thread at any time and is able to connect on one thread, to use the connection on another thread, and to disconnect on a third thread.

The connection pool is maintained by the Driver Manager. Connections are drawn from the pool when the application calls `SQLConnect` or `SQLDriverConnect`, and are returned to the pool when the application calls `SQLDisconnect`. The size of the pool grows dynamically, based on the requested resource allocations. It shrinks based on the inactivity timeout: If a connection is inactive for a period of time (it has not been used in a connection), it is removed from the pool. The size of the pool is limited only by memory constraints and limits on the server.

The Driver Manager determines whether a specific connection in a pool should be used according to the arguments passed in `SQLConnect` or `SQLDriverConnect`, and according to the connection attributes set after the connection was allocated.

When the Driver Manager is pooling connections, it needs to be able to determine if a connection is still working before handing out the connection. Otherwise, the Driver Manager keeps on hanging out the dead connection to the application whenever a transient network failure occurs. A new connection attribute has been defined in ODBC 3.x: `SQL_ATTR_CONNECTION_DEAD`. This is a read-only connection attribute that returns either `SQL_CD_TRUE` or `SQL_CD_FALSE`. The value `SQL_CD_TRUE` means that the connection has been lost, while the value `SQL_CD_FALSE` means that the connection is still active. (Drivers conforming to earlier versions of ODBC can also support this attribute.)

A driver must implement this option efficiently or it will impair the connection pooling performance. Specifically, a call to get this connection attribute should not cause a round trip to the server. Instead, a driver should just return the last known state of the connection. The connection is dead if the last trip to the server failed, and not dead if the last trip succeeded.

**Remarks**

If a connection has been lost (reported via `SQL_ATTR_CONNECTION_DEAD`), the ODBC Driver Manager will destroy that connection by calling `SQLDisconnect` in the driver. New connection requests might not find a usable connection in the pool. Eventually the Driver Manager might make a new connection, assuming the pool is empty.

To use a connection pool, an application performs the following steps:

1. Enables connection pooling by calling `SQLSetEnvAttr` to set the `SQL_ATTR_CONNECTION_POOLING` environment attribute to...
This call must be made before the application allocates the shared environment for which connection pooling is to be enabled. The environment handle in the call to SQLSetEnvAttr should be set to null, which makes SQL_ATTR_CONNECTION_POOLING a process-level attribute. If the attribute is set to SQL_CP_ONE_PER_DRIVER, a single connection pool is supported for each driver. If an application works with many drivers and few environments, this might be more efficient because fewer comparisons may be required. If set to SQL_CP_ONE_PER_HENV, a single connection pool is supported for each environment. If an application works with many environments and few drivers, this might be more efficient because fewer comparisons may be required. Connection pooling is disabled by setting SQL_ATTR_CONNECTION_POOLING to SQL_CP_OFF.

2. Allocates an environment by calling SQLAllocHandle with the HandleType argument set to SQL_HANDLE_ENV. The environment allocated by this call will be an implicit shared environment because connection pooling has been enabled. The environment to be used is not determined, however, until SQLAllocHandle with a HandleType of SQL_HANDLE_DBC is called on this environment.

3. Allocates a connection by calling SQLAllocHandle with InputHandle set to SQL_HANDLE_DBC, and the InputHandle set to the environment handle allocated for connection pooling. The Driver Manager attempts to find an existing environment that matches the environment attributes set by the application. If no such environment exists, one is created, with a reference count (maintained by the Driver Manager) of 1. If a matching shared environment is found, the environment is returned to the application and its reference count is incremented. (The actual connection to be used is not determined by the Driver Manager until SQLConnect or SQLDriverConnect is called.)

4. Calls SQLConnect or SQLDriverConnect to make the connection. The Driver Manager uses the connection options in the call to SQLConnect (or the connection keywords in the call to SQLDriverConnect) and the connection attributes set after connection allocation to determine which connection in the pool should be used.

   Note
   How a requested connection is matched to a pooled connection is determined by the SQL_ATTR_CP_MATCH environment attribute. For more information, see SQLSetEnvAttr.

   ODBC applications using connection pooling should call CoInitializeEx during application initialization and CoUninitialize when the application closes.

5. Calls SQLDisconnect when done with the connection. The connection is returned to the connection pool and becomes available for reuse.

   For an in-depth discussion, see Pooling in the Microsoft Data Access Components.

Connection Pooling Considerations

Performing any of the following actions using a SQL command (rather than through the ODBC API) can affect the connection's state and cause unexpected problems when connection pooling is active:

- Opening a connection and changing the default database.
- Using the SET statement to change any configurable options (including SET ROWCOUNT, ANSI_NULL, IMPLICIT_TRANSACTIONS, SHOWPLAN, STATISTICS, TEXTSIZE, and DATEFORMAT).
- Creating temporary tables and stored procedures.

If any of these actions are performed outside of the ODBC API, the next person who uses the connection will automatically inherit the previous settings, tables, or procedures.

   Note
   Do not expect certain settings to be present in the connection state. You should always set the connection state in your application and ensure that the application removes any unused connection pooling settings.

Driver–Aware Connection Pooling

Beginning in Windows 8, an ODBC driver can use connections in the pool more efficiently. For more information, see Driver-Aware Connection Pooling.

See Also

Connecting to a Data Source or Driver
Developing an ODBC Driver
Pooling in the Microsoft Data Access Components

Driver–Aware Connection Pooling

Driver aware connection pooling is a new feature of the Driver Manager in Windows 8. Driver aware connection pooling allows driver writers to customize the connection pooling behavior in their ODBC driver.
Driver awareness connection pooling is not supported with cursor library. An application will receive an error message if it attempts to enable cursor library via SQLSetConnectAttr, when driver aware connection pooling is enabled.

Driver aware connection pooling addresses the following problems related to Driver Manager connection pooling:

**Pool Fragmentation** The Driver Manager will only return a connection from the pool if it is an exact match with the connection string of a new connection request. One reason for the Driver Manager to require an exact match is that the Driver Manager does not understand every driver-specific connection string keyword and its value. However, some connection string keyword values (such as the name of the database) may not require an exact match, since the driver can change the database in less than the time needed to open a new connection (the exact time difference depends on the data source). And, differences in some connection attributes (such as SQL_ATTR_CURRENT_CATALOG) can take more time to change than differences in other attributes (such as SQL_ATTR_LOGIN_TIMEOUT). This, too, can prevent the Driver Manager from using the lowest-cost, reusable connection from the pool. When a driver has to create many new connections, an application's performance can decrease and the data source scalability can decrease. Pool fragmentation can be reduced with driver-aware connection pooling because a driver can better estimate the cost of reusing a connection in the pool for a connection request.

**No consideration of application preference** Some data sources can efficiently open new connections (compared to resetting some attributes), so, an application may prefer to open a new connection instead of trying to reuse a slightly mismatched connection from the pool and reset some values (although this may be slower during the connection pool initialization phrase). But some applications may keep the server load smaller and open fewer connections, although there may be a bigger cost to fix the mismatches for correct behavior. Without driver-aware connection pooling, you cannot specify this kind of preference effectively, because the Driver Manager does not recognize all driver-specific connection attributes. Driver-aware connection pooling allows a driver to obtain the user preference (with a driver-specific attribute of SQLSetConnectAttr) so that it can better estimate the cost of reusing a connection from the pool based on a user's preference.

For more information about driver-aware connection pooling, see Developing Connection-Pool Awareness in an ODBC Driver.

### Determining Driver Support

Driver-aware connection pooling is an optional feature that a driver may not support. To determine if a driver supports it, use the SQL_DRIVER_AWARE_POOLING_SUPPORTED InfoType of SQLGetInfo.

### How to Enable Driver–Aware Connection Pooling

An application can use a driver's connection-pooling awareness by setting the SQL_ATTR_CONNECTION_POOLING attribute to SQL_CP_DRIVER_AWARE with SQLSetEnvAttr. If a driver does not support connection-pool awareness, Driver Manager connection pooling will be used (same as if SQL_CP_ONE_PER_HENV had been specified, instead of SQL_CP_DRIVER_AWARE). ODBC 2.x and 3.x applications can enable this feature.

### See Also

- Developing an ODBC Driver

### Disconnecting from a Data Source or Driver

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. It returns an error if a transaction is in process.

After disconnecting, the application can call SQLFreeHandle to free the connection. After freeing the connection, it is an application programming error to use the connection's handle in a call to an ODBC function; doing so has undefined but probably fatal consequences. When SQLFreeHandle is called, the driver releases the structure used to store information about the connection.

The application also can reuse the connection, either to connect to a different data source or 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.

### Driver Manager's Role in the Connection Process

Remember that applications do not call driver functions directly. Instead, they call Driver Manager functions with the same name and the Driver Manager calls the driver functions. Usually, this happens almost immediately. For example, the application calls SQLExecute in the Driver Manager and after a few error checks, the Driver Manager calls SQLExecute in the driver.

The connection process is different. When the application calls SQLAllocHandle with the SQL_HANDLE_ENV and SQL_HANDLE_DBC options, the function allocates handles only in the Driver Manager. The Driver Manager does not call this function in the driver because it does not know which driver to call. Similarly, if the application passes the handle of an unconnected connection to SQLSetConnectAttr or SQLGetConnectAttr, only the Driver Manager
executes the function. It stores or gets the attribute value from its connection handle and returns SQLSTATE 08003 (Connection not open) when getting a value for an attribute that has not been set and for which ODBC does not define a default value.

When the application calls SQLConnect, SQLDriverConnect, or SQLBrowseConnect, the Driver Manager first determines which driver to use. It then checks to determine whether a driver is currently loaded on the connection:

- If no driver is loaded on the connection, the Driver Manager checks whether the specified driver is loaded on another connection in the same environment. If not, the Driver Manager loads the driver on the connection and calls SQLAllocHandle in the driver with the SQL_HANDLE_ENV option.

  The Driver Manager then calls SQLAllocHandle in the driver with the SQL_HANDLE_DBC option, whether or not it was just loaded. If the application set any connection attributes, the Driver Manager calls SQLSetConnectAttr in the driver; if an error occurs, the Driver Manager’s connection function returns SQLSTATE IM006 (Driver’s SQLSetConnectAttr failed). Finally, the Driver Manager calls the connection function in the driver.

- If the specified driver is loaded on the connection, the Driver Manager calls only the connection function in the driver. In this case, the driver must make sure that all connection attributes maintain their current settings.

- If a different driver is loaded on the connection, the Driver Manager calls SQLFreeHandle in the driver to free the connection. If there are no other connections that use the driver, the Driver Manager calls SQLFreeHandle in the driver to free the environment and unload the driver. The Driver Manager then performs the same operations as when a driver is not loaded on the connection.

The Driver Manager will lock the environment handle (henv) before calling a driver’s SQLAllocHandle and SQLFreeHandle when HandleType is set to SQL_HANDLE_DBC.

When the application calls SQLDisconnect, the Driver Manager calls SQLDisconnect in the driver. However, it leaves the driver loaded in case the application reconnects to the driver. When the application calls SQLFreeHandle with the SQL_HANDLE_DBC option, the Driver Manager calls SQLFreeHandle in the driver. If the driver is not used by any other connections, the Driver Manager then calls SQLFreeHandle in the driver with the SQL_HANDLE_ENV option and unloads the driver.

© 2016 Microsoft

Catalog Functions

All databases have a structure that outlines how data will be stored in the database. For example, a simple sales order database might have the structure shown in the following illustration, in which the ID columns are used to link the tables.

This structure, along with other information such as privileges, is stored in a set of system tables called the database’s catalog, which is also known as a data dictionary.

An application can discover this structure through calls to the catalog functions. The catalog functions return information in result sets and are usually implemented through SELECT statements against the 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.

This section contains the following topics:

- Uses of Catalog Data
- Catalog Functions in ODBC

© 2016 Microsoft

Uses of Catalog Data

Applications use catalog data in a variety of ways. Here are some common uses:

- Constructing SQL statements at run time. Vertical applications, such as an order entry application, contain hard-coded SQL statements.
The tables and columns that are used by the application are fixed ahead of time, as are the statements that access these tables. For example, an order entry application usually contains a single, parameterized `INSERT` statement for adding new orders to the system.

Generic applications, such as a spreadsheet program that uses ODBC to retrieve data, often construct SQL statements at run time based on input from the user. Such an application could require the user to type the names of the tables and columns to use. However, it would be easier for the user if the application displayed lists of tables and columns from which the user could make selections. To build these lists, the application would call the `SQLTables` and `SQLColumns` catalog functions.

- **Constructing SQL statements during development.** Application development environments typically allow the programmer to create database queries while developing a program. The queries are then hard-coded in the application being built. Such environments could also use `SQLTables` and `SQLColumns` to create lists from which the programmer could make selections. These environments might also use `SQLPrimaryKeys` and `SQLForeignKeys` to automatically determine and show relationships between selected tables, and use `SQLStatistics` to determine and highlight indexed fields so the programmer can create efficient queries.

- **Constructing cursors.** An application, driver, or middleware that provides a scrollable cursor engine could use `SQLSpecialColumns` to determine which column or columns uniquely identify a row. The program could build a keyset containing the values of these columns for each row that has been fetched. When the application scrolls back to the row, it would then use these values to fetch the most recent data for the row. For more information about scrollable cursors and keysets, see Scrollable Cursors.

---

**Catalog Functions in ODBC**

ODBC contains the following catalog functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLTables</td>
<td>Returns a list of catalogs, schemas, tables, or table types in the data source.</td>
</tr>
<tr>
<td>SQLColumns</td>
<td>Returns a list of columns in one or more tables.</td>
</tr>
<tr>
<td>SQLStatistics</td>
<td>Returns a list of statistics about a single table. Also returns a list of indexes associated with that table.</td>
</tr>
<tr>
<td>SQLSpecialColumns</td>
<td>Returns a list of columns that uniquely identifies a row in a single table. Also returns a list of columns in that table that are automatically updated.</td>
</tr>
<tr>
<td>SQLPrimaryKeys</td>
<td>Returns a list of columns that compose the primary key of a single table.</td>
</tr>
<tr>
<td>SQLForeignKeys</td>
<td>Returns a list of foreign keys in a single table or a list of foreign keys in other tables that refer to a single table.</td>
</tr>
<tr>
<td>SQLTablePrivileges</td>
<td>Returns a list of privileges associated with one or more tables.</td>
</tr>
<tr>
<td>SQLColumnPrivileges</td>
<td>Returns a list of privileges associated with one or more columns in a single table.</td>
</tr>
<tr>
<td>SQLProcedures</td>
<td>Returns a list of procedures in the data source.</td>
</tr>
<tr>
<td>SQLProcedureColumns</td>
<td>Returns a list of input and output parameters, the return value, and the columns in the result set of a single procedure.</td>
</tr>
<tr>
<td>SQLGetTypeInfo</td>
<td>Returns a list of the SQL data types supported by the data source. These data types are generally used in <code>CREATE TABLE</code> and <code>ALTER TABLE</code> statements.</td>
</tr>
</tbody>
</table>

Because `SQLTables`, `SQLColumns`, `SQLStatistics`, and `SQLSpecialColumns` conform to the Open Group CLI, and `SQLGetTypeInfo` conforms to the ISO 92 CLI, they are implemented by most drivers. The remaining catalog functions are in the ODBC conformance level.

This section contains the following topics.

- Data Returned by Catalog Functions
- Arguments in Catalog Functions
- Schema Views

---

**Data Returned by Catalog Functions**

Each catalog function returns data as a result set. This result set is no different from any other result set. It is usually generated by a predefined,
parameterized SELECT statement that is hard-coded in the driver or stored in a procedure in the data source. For information about how to retrieve data from
a result set, see Was a Result Set Created?

The result set for each catalog function is described in the reference entry for that function. In addition to the listed columns, the result set can contain driver-specific columns after the last predefined column. These columns (if any) are described in the driver documentation.

Applications should bind driver-specific columns relative to the end of the result set. That is, they should calculate the number of a driver-specific column as the number of the last column — retrieved with SQLNumResultCols — less the number of columns that occur after the required column. This saves having to change the application when new columns are added to the result set in future versions of ODBC or the driver. For this scheme to work, drivers must add new driver-specific columns before old driver-specific columns so that column numbers do not change relative to the end of the result set.

Identifiers that are returned in the result set are not quoted, even if they contain special characters. For example, suppose the identifier quote character (which is driver-specific and returned through SQLGetInfo) is a double quotation mark (") and the Accounts Payable table contains a column named Customer Name. In the row returned by SQLColumns for this column, the value of the TABLE_NAME column is Accounts Payable, not "Accounts Payable", and the value of the COLUMN_NAME column is Customer Name, not "Customer Name". To retrieve the names of customers in the Accounts Payable table, the application would quote these names:

```sql
SELECT "Customer Name" FROM "Accounts Payable"
```

For more information, see Quoted Identifiers.

The catalog functions are based on an SQL-like authorization model in which a connection is made based on a user name and password, and only data for which the user has a privilege is returned. Password protection of individual files, which does not fit into this model, is driver-defined.

The result sets returned by the catalog functions are almost never updatable, and applications should not expect to be able to change the structure of the database by changing the data in these result sets.

© 2016 Microsoft

Arguments in Catalog Functions

All catalog functions accept arguments with which an application can restrict the scope of the data returned. For example, the first and second calls to SQLTables in the following code return a result set containing information about all tables, while the third call returns information about the Orders table:

```c
SQLTables(hstmt1, NULL, 0, NULL, NULL, 0, NULL, 0);
SQLTables(hstmt2, NULL, 0, NULL, 0, "", SQL_NTS, NULL, 0);
SQLTables(hstmt3, NULL, 0, NULL, 0, "Orders", SQL_NTS, NULL, 0);
```

Catalog function string arguments fall into four different types: ordinary argument (OA), pattern value argument (PV), identifier argument (ID), and value list argument (VL). Most string arguments can be of one of two different types, depending on the value of the SQL_ATTR_METADATA_ID statement attribute. The following table lists the arguments for each catalog function and describes the type of the argument for an SQL_TRUE or SQL_FALSE value of SQL_ATTR_METADATA_ID.

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLColumnPrivileges</td>
<td>CatalogName, SchemaName, TableName, ColumnName</td>
</tr>
<tr>
<td>SQLColumns</td>
<td>CatalogName, SchemaName, TableName, ColumnName</td>
</tr>
<tr>
<td>SQLForeignKeys</td>
<td>PKCatalogName, PKSchemaName, PKTableName, PKCatalogName, FKName</td>
</tr>
<tr>
<td>SQLPrimaryKeys</td>
<td>CatalogName, SchemaName, TableName</td>
</tr>
<tr>
<td>SQLProcedureColumns</td>
<td>CatalogName, SchemaName, ProcName, ColumnName</td>
</tr>
<tr>
<td>SQLProcedures</td>
<td>CatalogName, SchemaName, ProcName</td>
</tr>
<tr>
<td>SQLSpecialColumns</td>
<td>CatalogName, SchemaName, TableName</td>
</tr>
<tr>
<td>SQLStatistics</td>
<td>CatalogName, SchemaName, TableName</td>
</tr>
<tr>
<td>SQLTablePrivileges</td>
<td>CatalogName, SchemaName, TableName</td>
</tr>
<tr>
<td>SQLTables</td>
<td>CatalogName, SchemaName, TableName, TableType</td>
</tr>
</tbody>
</table>
This section contains the following topics.

- Ordinary Arguments
- Pattern Value Arguments
- Identifier Arguments
- Value List Arguments

© 2016 Microsoft

Ordinary Arguments

When a catalog function string argument is an ordinary argument, it is treated as a literal string. An ordinary argument accepts neither a string search pattern nor a list of values. The case of an ordinary argument is significant, and quote characters in the string are taken literally. These arguments are treated as ordinary arguments if the SQL_ATTR_METADATA_ID statement attribute is set to SQL_FALSE; they are treated as identifier arguments instead if this attribute is set to SQL_TRUE.

If an ordinary argument is set to a null pointer and the argument is a required argument, the function returns SQL_ERROR and SQLSTATE HY009 (Invalid use of null pointer). If an ordinary argument is set to a null pointer and the argument is not a required argument, the argument’s behavior is driver-dependent. The required arguments are listed in the following table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Required arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLColumnPrivileges</td>
<td>TableName</td>
</tr>
<tr>
<td>SQLForeignKeys</td>
<td>PKTableName, FKTableName</td>
</tr>
<tr>
<td>SQLPrimaryKeys</td>
<td>TableName</td>
</tr>
<tr>
<td>SQLSpecialColumns</td>
<td>TableName</td>
</tr>
<tr>
<td>SQLStatistics</td>
<td>TableName</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

Pattern Value Arguments

Some arguments in the catalog functions, such as the TableName argument in SQLTables, accept search patterns. These arguments accept search patterns if the SQL_ATTR_METADATA_ID statement attribute is set to SQL_FALSE; they are identifier arguments that do not accept a search pattern if this attribute is set to SQL_TRUE.

The search pattern characters are:

- An underscore (_), which represents any single character.
- A percent sign (%), which represents any sequence of zero or more characters.
- An escape character, which is driver-specific and is used to include underscores, percent signs, and the escape character as literals. If the escape character precedes a non-special character, the escape character has no special meaning. If the escape character precedes a special character, it escapes the special character. For example, "\a" would be treated as two characters, "\" and "a", but "\%" would be treated as the non-special single character "%".

The escape character is retrieved with the SQL_SEARCH_PATTERN_ESCAPE option in SQLGetInfo. It must precede any underscore, percent sign, or escape character in an argument that accepts search patterns to include that character as a literal. Examples are shown in the following table.

<table>
<thead>
<tr>
<th>Search pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%A</td>
<td>All identifiers containing the letter A</td>
</tr>
<tr>
<td>ABC_</td>
<td>All four character identifiers starting with ABC</td>
</tr>
<tr>
<td>ABC_</td>
<td>The identifier ABC_, assuming the escape character is a backslash ()</td>
</tr>
<tr>
<td>\%</td>
<td>All identifiers starting with a backslash (), assuming the escape character is a backslash</td>
</tr>
</tbody>
</table>

Special care must be taken to escape search pattern characters in arguments that accept search patterns. This is particularly true for the underscore.
character, which is commonly used in identifiers. A common mistake in applications is to retrieve a value from one catalog function and pass that value to a search pattern argument in another catalog function. For example, suppose an application retrieves the table name MY_TABLE from the result set for SQLTables and passes this to SQLColumns to retrieve a list of columns in MY_TABLE. Instead of getting the columns for MY_TABLE, the application will get the columns for all the tables that match the search pattern MY_TABLE, such as MY_TABLE, MY1TABLE, MY2TABLE, and so on.

Note

ODBC 2.x drivers do not support search patterns in the CatalogName argument in SQLTables. ODBC 3.x drivers accept search patterns in this argument if the SQL_ATTR_ODBC_VERSION environment attribute is set to SQL_OV_ODBC3; they do not accept search patterns in this argument if it is set to SQL_OV_ODBC2.

Passing a null pointer to a search pattern argument does not constrain the search for that argument; that is, a null pointer and the search pattern % (any characters) are equivalent. However, a zero-length search pattern — that is, a valid pointer to a string of length zero — matches only the empty string ("").

© 2016 Microsoft

Identifier Arguments

If a string in an identifier argument is quoted, the driver removes leading and trailing blanks and treats literally the string within the quotation marks. If the string is not quoted, the driver removes trailing blanks and folds the string to uppercase. Setting an identifier argument to a null pointer returns SQL_ERROR and SQLSTATE HY009 (Invalid use of null pointer), unless the argument is a catalog name and catalogs are not supported.

These arguments are treated as identifier arguments if the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE. In this case, the underscore (_) and the percent sign (%) will be treated as the actual character, not as a search pattern character. These arguments are treated as either an ordinary argument or a pattern argument, depending on the argument, if this attribute is set to SQL_FALSE.

Although identifiers containing special characters must be quoted in SQL statements, they must not be quoted when passed as catalog function arguments, because quote characters passed to catalog functions are interpreted literally. For example, suppose the identifier quote character (which is driver-specific and returned through SQLGetInfo) is a double quotation mark (""). The first call to SQLTables returns a result set containing information about the Accounts Payable table, while the second call returns information about the "Accounts Payable" table, which is probably not what was intended.

Quoted identifiers are used to distinguish a true column name from a pseudo-column of the same name, such as ROWID in Oracle. If "ROWID" is passed in an argument of a catalog function, the function will work with the ROWID pseudo-column if it exists. If the pseudo-column does not exist, the function will work with the "ROWID" column. If ROWID is passed in an argument of a catalog function, the function will work with the ROWID column.

For more information about quoted identifiers, see Quoted Identifiers.

© 2016 Microsoft

Value List Arguments

A value list argument consists of a list of comma-separated values to be used for matching. There is only one value list argument in the ODBC catalog functions: the TableType argument in SQLTables. Setting TableType to a null pointer is the same as if it is set to SQL_ALL_TABLE_TYPES, which enumerates all possible members of the value list. This argument is not affected by the SQL_ATTR_METADATA_ID statement attribute. For more information, see the SQLTables function description.

© 2016 Microsoft

Schema Views

An application can retrieve metadata information from the DBMS either by calling ODBC catalog functions or by using INFORMATION_SCHEMA views. The views are defined by the ANSI SQL-92 standard.

If supported by the DBMS and the driver, the INFORMATION_SCHEMA views provide a more powerful and comprehensive means of retrieving metadata than the ODBC catalog functions provide. An application can execute its own custom SELECT statement against one of these views, can join views, or can perform a union on views. While offering greater utility and a wider range of metadata, INFORMATION_SCHEMA views are not often supported by the DBMS. This might change as more DBMSs and drivers achieve compliance with SQL-92.

To determine which views are supported, an application calls SQLGetInfo with the SQL_INFO_SCHEMA_VIEWS option. To retrieve metadata from a supported view, the application executes a SELECT statement that specifies the schema information required.
ODBC applications perform almost all database access by executing SQL statements. The form of these statements — hard-coded or constructed at run time, interoperable or data source–specific, and so on — depends on the needs of the application.

This section contains the following topics.

- Constructing SQL Statements
- Interoperability of SQL Statements
- Escape Sequences in ODBC

## Constructing SQL Statements

SQL statements can be constructed in one of three ways: hard-coded during development, constructed at run time, or entered directly by the user.

This section contains the following topics.

- Hard-Coded SQL Statements
- SQL Statements Constructed at Run Time
- SQL Statements Entered by the User

### Hard-Coded SQL Statements

Applications that perform a fixed task usually contain hard-coded SQL statements. For example, an order entry system might use the following call to list open sales orders:

```sql
SQLExecDirect(hstmt, "SELECT OrderID FROM Orders WHERE Status = 'OPEN'", SQL_NTS);
```

There are several advantages to hard-coded SQL statements: They can be tested when the application is written; they are simpler to implement than statements constructed at run time; and they simplify the application.

Using statement parameters and preparing statements provide even better ways to use hard-coded SQL statements. For example, suppose the Parts table contains the PartID, Description, and Price columns. One way to insert a new row into this table would be to construct and execute an INSERT statement:

```sql
#define DESC_LEN 51
#define STATEMENT_LEN 51
SQLUINTEGER PartID;
SQLCHAR Desc[DESC_LEN], Statement[STATEMENT_LEN];
SQLREAL Price;

// Set part ID, description, and price.
GetNewValues(&PartID, Desc, &Price);

// Build INSERT statement.
sprintf_s(Statement, 100, "INSERT INTO Parts (PartID, Description, Price) "
    "VALUES (%d, '%s', %f)", PartID, Desc, Price);

// Execute the statement.
SQLExecDirect(hstmt, Statement, SQL_NTS);
```
An even better way is to use a hard-coded, parameterized statement. This has two advantages over a statement with hard-coded data values. First, it is easier to construct a parameterized statement because the data values can be sent in their native types, such as integers and floating-point numbers, rather than converting them to strings. Second, such a statement can be used more than once simply by changing the parameter values and reexecuting it; there is no need to rebuild it.

Assuming this statement is to be executed more than once, it can be prepared for even greater efficiency:

Perhaps the most efficient way to use the statement is to construct a procedure containing the statement, as shown in the following code example. Because the procedure is constructed at development time and stored on the data source, it does not need to be prepared at run time. A drawback of this method is that the syntax for creating procedures is DBMS-specific and procedures must be constructed separately for each DBMS on which the application is to run.

```c
#define DESC_LEN 51

SQLCHAR * Statement = "INSERT INTO Parts (PartID, Description, Price) "
    "VALUES (?, ?, ?)";
SQLUINTEGER PartID;
SQLCHAR Desc[DESC_LEN];
SQLREAL Price;
SQLINTEGER PartIDInd = 0, DescLenOrInd = SQL_NTS, PriceInd = 0;

// Bind the parameters.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
    &PartID, 0, &PartIDInd);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, DESC_LEN - 1, 0,
    Desc, sizeof(Desc), &DescLenOrInd);
SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,
    &Price, 0, &PriceInd);

// Set part ID, description, and price.
GetNewValues(&PartID, Desc, &Price);

// Execute the statement.
SQLExecDirect(hstmt, Statement, SQL_NTS);
```
SQL Statements Constructed at Run Time

Applications that perform ad hoc analysis commonly build SQL statements at run time. For example, a spreadsheet might allow a user to select columns from which to retrieve data:

```c
#include <windows.h>
#include <stdio.h>
#include <sqltypes.h>

int main() {
    SQLCHAR *Statement = 0, *TableName = 0;
    SQLCHAR **TableNameArray, **ColumnNamesArray = 0;
    BOOL *ColumnSelectedArray = 0;
    SQLSMALLINT i = 0, NumColumns = 0;

    // Use SQLTables to build a list of tables (TableNameArray[]). Let the
    // user select a table and store the selected table in TableName.
    // Use SQLColumns to build a list of the columns in the selected table
    // (ColumnNamesArray). Set NumColumns to the number of columns in the
    // table. Let the user select one or more columns and flag these columns
    // in ColumnSelectedArray[].
    // Build a SELECT statement from the selected columns.
    CommaNeeded = FALSE;
    Statement = (SQLCHAR*)malloc(8);
    strcat_s((char*)Statement, 8, "SELECT ");
    for (i = 0 ; i = NumColumns ; i++) {
        if (ColumnSelectedArray[i]) {
            if (CommaNeeded)
                strcat_s((char*)Statement, sizeof(Statement), ",");
            else
                CommaNeeded = TRUE;
            strcat_s((char*)Statement, sizeof(Statement), (char*)ColumnNamesArray[i]);
        }
    }
    strcat_s((char*)Statement, 15, " FROM ");
    // strcat_s((char*)Statement, 100, (char*)TableName);
    // Execute the statement. It will be executed once, do not prepare it.
    // SQLExecDirect(hstmt, Statement, SQL_NTS);
}
```

Another class of applications that commonly constructs SQL statements at run time are application development environments. However, the statements they construct are hard-coded in the application they are building, where they can usually be optimized and tested.

Applications that construct SQL statements at run time can provide tremendous flexibility to the user. As can be seen from the preceding example, which did not even support such common operations as `WHERE` clauses, `ORDER BY` clauses, or joins, constructing SQL statements at run time is vastly more complex than hard-coding statements. Furthermore, testing such applications is problematic because they can construct an arbitrary number of SQL statements.

A potential disadvantage of constructing SQL statements at run time is that it takes far more time to construct a statement than use a hard-coded statement. Fortunately, this is rarely a concern. Such applications tend to be user-interface intensive, and the time the application spends constructing SQL statements is generally small compared to the time the user spends entering criteria.

SQL Statements Entered by the User

Applications that perform ad hoc analysis also commonly allow the user to enter SQL statements directly. For example:

```c
SQLCHAR * Statement, SqlState[6], Msg[SQL_MAX_MESSAGE_LENGTH];
SQLSMALLINT rc1, rc2;
```
This approach simplifies application coding; the application relies on the user to build the SQL statement and on the data source to check the statement's validity. Because it's difficult to write a graphical user interface that adequately exposes the intricacies of SQL, simply asking the user to enter the SQL statement text may be a preferable alternative. However, this requires the user to know not only SQL but also the schema of the data source being queried. Some applications provide a graphical user interface by which the user can create a basic SQL statement and also provide a text interface with which the user can modify it.

Interoperability of SQL Statements

Like the rest of an application, SQL statements can be interoperable or DBMS-specific. And like the rest of the application, the choice of how interoperable SQL statements need to be depends on the type of application. Custom applications are less likely to use interoperable SQL statements because they are usually designed to exploit the capabilities of one or possibly two DBMSs. Generic applications use interoperable SQL statements because they are designed to work with a variety of DBMSs. And vertical applications usually fall somewhere in between, demanding a certain level of functionality but otherwise using interoperable SQL statements.

This section contains the following topics.

- Choosing an SQL Grammar
- Constructing Interoperable SQL Statements

Choosing an SQL Grammar

The first decision to make when constructing SQL statements is which grammar to use. In addition to the grammars available from the various standards bodies, such as Open Group, ANSI, and ISO, virtually every DBMS vendor defines its own grammar, each of which varies slightly from the standard.

Appendix C: SQL Grammar, describes the minimum SQL grammar that all ODBC drivers must support. This grammar is a subset of the Entry level of SQL-92. Drivers may support additional grammar to conform to the Intermediate, Full, or FIPS 127-2 Transitional levels defined by SQL-92. For more information, see SQL Minimum Grammar in Appendix C: SQL Grammar, and SQL-92.

Appendix C also defines escape sequences containing standard grammar for commonly available language features, such as outer joins, that are not covered by the SQL-92 grammar. For more information, see ODBC Escape Sequences in Appendix C: SQL Grammar, and Escape Sequences, later in this section.

The grammar that is chosen affects how the driver processes the statement. Drivers must modify SQL-92 SQL and the ODBC-defined escape sequences to DBMS-specific SQL. Because most SQL grammars are based on one or more of the various standards, most drivers do little or no work to meet this requirement. It often consists only of searching for the escape sequences defined by ODBC and replacing them with DBMS-specific grammar. When a driver encounters grammar it does not recognize, it assumes the grammar is DBMS-specific and passes the SQL statement without modification to the data source for execution.

Therefore, there are really two choices of grammar to use: the SQL-92 grammar (and the ODBC escape sequences) and a DBMS-specific grammar. Of the two, only the SQL-92 grammar is interoperable, so all interoperable applications should use it. Applications that are not interoperable can use the SQL-92 grammar or a DBMS-specific grammar. DBMS-specific grammars have two advantages: They can exploit any features not covered by SQL-92, and they are marginally faster because the driver does not have to modify them. The latter feature can be partially enforced by setting the SQL_ATTR_NOSCAN statement attribute, which stops the driver from searching for and replacing escape sequences.

If the SQL-92 grammar is used, the application can discover how it is modified by the driver by calling SQLNativeSql. This is often useful when debugging applications. SQLNativeSql accepts an SQL statement and returns it after the driver has modified it. Because this function is in the Core interface conformance level, it is supported by all drivers.

Constructing Interoperable SQL Statements
As mentioned in the previous sections, interoperable applications should use the ODBC SQL grammar. Beyond using this grammar, however, a number of additional problems are faced by interoperable applications. For example, what does an application do if it wants to use a feature, such as outer joins, that is not supported by all data sources?

At this point, the application writer must make some decisions about which language features are required and which are optional. In most cases, if a particular driver does not support a feature required by the application, the application simply refuses to run with that driver. However, if the feature is optional, the application can work around the feature. For example, it might disable those parts of the interface that allow the user to use the feature.

To determine which features are supported, applications start by calling SQLGetInfo with the SQL_SQL_CONFORMANCE option. The SQL conformance level gives the application a broad view of which SQL is supported. To refine this view, the application calls SQLGetInfo with any of a number of other options. For a complete list of these options, see the SQLGetInfo function description. Finally, SQLGetTypeInfo returns information about the data types supported by the data source. The following sections list a number of possible factors that applications should watch for when constructing interoperable SQL statements.

This section contains the following topics.

- Catalog and Schema Usage
- Catalog Position
- Quoted Identifiers
- Identifier Case
- Escape Sequences
- Literal Prefixes and Suffixes
- Parameter Markers in Procedure Calls
- DDL Statements

Catalog and Schema Usage

Data sources do not necessarily support catalog and schema names as object name identifiers in all SQL statements. Data sources might support catalog and schema names in one or more of the following classes of SQL statements: Data Manipulation Language (DML) statements, procedure calls, table definition statements, index definition statements, and privilege definition statements. To determine the classes of SQL statements in which catalog and schema names can be used, an application calls SQLGetInfo with the SQL_CATALOG_USAGE and SQL_SCHEMA_USAGE options.

Catalog Position

The position of a catalog name in an identifier and how it is separated from the rest of the identifier varies from data source to data source. For example, in an Xbase data source, the catalog name is a directory and, in Microsoft® Windows®, is separated from the table name (which is a file name) by a backslash (\). The following illustration demonstrates this condition.

```sql
\DBASE\SALES\CORP\PARTS.DBF
```

In a SQL Server data source, the catalog is a database and is separated from the schema and table names by a period (.)

```sql
Sales CORPORATE Parts
```

In an Oracle data source, the catalog is also the database but follows the table name and is separated from the schema and table names by an at sign (@).

```sql
Corporate Parts@Sales
```

To determine the catalog separator and the location of the catalog name, an application calls SQLGetInfo with the SQL_CATALOG_NAME_SEPARATOR and
SQL_CATALOG_LOCATION options. Interoperable applications should construct identifiers according to these values.

When quoting identifiers that contain more than one part, applications must be careful to quote each part separately and not quote the character that separates the identifiers. For example, the following statement to select all of the rows and columns of an Xbase table quotes the catalog (\XBASE\SALES\CORP) and table (Parts.dbf) names, but not the catalog separator (\):

```
SELECT * FROM "\XBASE\SALES\CORP"."PARTS.DBF"
```

The following statement to select all of the rows and columns of an Oracle table quotes the catalog (Sales), schema (Corporate), and table (Parts) names, but not the catalog (@) or schema (.) separators:

```
SELECT * FROM "Corporate"."Parts"@"Sales"
```

For information about quoting identifiers, see the next section, Quoted Identifiers.

---

**Quoted Identifiers**

In an SQL statement, identifiers containing special characters or match keywords must be enclosed in *identifier quote characters*; identifiers enclosed in such characters are known as *quoted identifiers* (also known as *delimited identifiers* in SQL-92). For example, the Accounts Payable identifier is quoted in the following `SELECT` statement:

```
SELECT * FROM "Accounts Payable"
```

The reason for quoting identifiers is to make the statement parseable. For example, if Accounts Payable was not quoted in the previous statement, the parser would assume there were two tables, Accounts and Payable, and return a syntax error that they were not separated by a comma. The identifier quote character is driver-specific and is retrieved with the `SQL_IDENTIFIER_QUOTE_CHAR` option in `SQLGetInfo`. The lists of special characters and of keywords are retrieved with the `SQL_SPECIAL_CHARACTERS` and `SQL_KEYWORDS` options in `SQLGetInfo`.

To be safe, interoperable applications often quote all identifiers except those for pseudo-columns, such as the ROWID column in Oracle. `SQLSpecialColumns` returns a list of pseudo-columns. Also, if there are application-specific restrictions on where special characters can appear in an object name, it is best for interoperable applications not to use special characters in those positions.

---

**Identifier Case**

In SQL statements and catalog function arguments, identifiers and quoted identifiers can be either case-sensitive or not, which an application can determine by calling `SQLGetInfo` with the `SQL_IDENTIFIER_CASE` and `SQL_QUOTED_IDENTIFIER_CASE` options.

Each of these options has four possible return values: one stating that the identifier or quoted identifier case is sensitive and three stating that it is not sensitive. The three values that are not case-sensitive further describe the case in which identifiers are stored in the system catalog. How identifiers are stored in the system catalog is relevant only for display purposes, such as when an application displays the results of a catalog function; it does not change the case-sensitivity of identifiers.

---

**Escape Sequences**

ODBC defines escape sequences containing standard grammar for date, time, timestamp, and datetime interval literals, scalar function calls, `LIKE` predicate escape characters, outer joins, and procedure calls. Interoperable applications should use these sequences whenever possible.

To determine if a driver supports the escape sequences for date, time, timestamp, or datetime interval literals, an application calls `SQLGetTypeInfo`. If the data source supports a date, time, timestamp, or datetime interval data type, it must also support the corresponding escape sequence. To determine whether the other escape sequences are supported, an application calls `SQLGetInfo`.

For more information, see Escape Sequences in ODBC, later in this section.
Literal Prefixes and Suffixes

In an SQL statement, a literal is a character representation of an actual data value. For example, in the following statement, ABC, FFFF, and 10 are literals:

```
SELECT CharCol, BinaryCol, IntegerCol FROM MyTable
 WHERE CharCol = 'ABC' AND BinaryCol = 0xFFFF AND IntegerCol = 10
```

Literals for some data types require special prefixes and suffixes. In the preceding example, the character literal (ABC) requires a single quotation mark (’) as both a prefix and a suffix, the binary literal (FFFF) requires the characters 0x as a prefix, and the integer literal (10) does not require a prefix or suffix.

For all data types except date, time, and timestamps, interoperable applications should use the values returned in the LITERAL_PREFIX and LITERAL_SUFFIX columns in the result set created by SQLGetTypeInfo. For date, time, timestamp, and datetime interval literals, interoperable applications should use the escape sequences discussed in the preceding section.

Parameter Markers in Procedure Calls

When calling procedures that accept parameters, interoperable applications should use parameter markers instead of literal parameter values. Some data sources do not support the use of literal parameter values in procedure calls. For more information about parameters, see Statement Parameters. For more information about calling procedures, see Procedure Calls, later in this section.

DDL Statements

Data Definition Language (DDL) statements vary tremendously among DBMSs. ODBC SQL defines statements for the most common data definition operations: creating and dropping tables, indexes, and views; altering tables; and granting and revoking privileges. All other DDL statements are data source–specific. Therefore, interoperable applications cannot perform some data definition operations. In general, this is not a problem, because such operations tend to be highly DBMS-specific and are best left to the proprietary database administration software shipped with most DBMSs or the setup program shipped with the driver.

Another problem in data definition is that data type names vary tremendously among DBMSs. Rather than defining standard data type names and forcing drivers to convert them to DBMS-specific names, SQLGetTypeInfo provides a way for applications to discover DBMS-specific data type names. Interoperable applications should use these names in SQL statements to create and alter tables; the names listed in Appendix C: SQL Grammar, and Appendix D: DataTypes, are examples only.

Escape Sequences in ODBC

A number of language features, such as outer joins and scalar function calls, are commonly implemented by DBMSs. However, the syntaxes for these features tend to be DBMS-specific, even when standard syntaxes are defined by the various standards bodies. Because of this, ODBC defines escape sequences that contain standard syntaxes for the following language features:

- Date, time, timestamp, and datetime interval literals
- Scalar functions such as numeric, string, and data type conversion functions
- LIKE predicate escape character
- Outer joins
- Procedure calls
The escape sequence used by ODBC is as follows:

(extension)

Remarks

The escape sequence is recognized and parsed by drivers, which replace the escape sequences with DBMS-specific grammar. For more information about escape sequence syntax, see ODBC Escape Sequences in Appendix C: SQL Grammar.

Note

In ODBC 2.x, this was the standard syntax of the escape sequence:  --(*vendor(vendor-name), product(product-name) extension *)--
In addition to this syntax, a shorthand syntax was defined of the form: {extension}
In ODBC 3.x, the long form of the escape sequence has been deprecated, and the shorthand form is used exclusively.

Because the escape sequences are mapped by the driver to DBMS-specific syntaxes, an application can use either the escape sequence or DBMS-specific syntax. However, applications that use the DBMS-specific syntax will not be interoperable. When using the escape sequence, applications should make sure that the SQL_ATTR_NOSCAN statement attribute is turned off, which it is by default. Otherwise, the escape sequence will be sent directly to the data source, where it will generally cause a syntax error.

Drivers support only those escape sequences that they can map to underlying language features. For example, if the data source does not support outer joins, neither will the driver. To determine which escape sequences are supported, an application calls SQLGetTypeInfo and SQLGetInfo. For more information, see the next section, Date, Time, and Timestamp Literals.

This section contains the following topics.

- Date, Time, and Timestamp Literals
- Scalar Function Calls
- LIKE Predicate Escape Character
- Outer Joins
- Procedure Calls

© 2016 Microsoft

Date, Time, and Timestamp Literals

The escape sequence for date, time, and timestamp literals is

{ -type 'value'}

where literal-type is one of the values listed in the following table.

<table>
<thead>
<tr>
<th>literal-type</th>
<th>Meaning</th>
<th>Format of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Date</td>
<td>yyyy-mm-dd</td>
</tr>
<tr>
<td>t</td>
<td>Time*</td>
<td>hh:mm:ss[1]</td>
</tr>
<tr>
<td>ts</td>
<td>Timestamp</td>
<td>yyyy-mm-dd hh:mm:ss[f...][1]</td>
</tr>
</tbody>
</table>

[1] The number of digits to the right of the decimal point in a time or timestamp interval literal containing a seconds component is dependent on the seconds precision, as contained in the SQL_DESC_PRECISION descriptor field. (For more information, see SQLSetDescField.)

For more information about the date, time, and timestamp escape sequences, see Date, Time, and Timestamp Escape Sequences in Appendix C: SQL Grammar.

For example, both of the following SQL statements update the open date of sales order 1023 in the Orders table. The first statement uses the escape sequence syntax. The second statement uses the Oracle Rdb native syntax for the DATE column and is not interoperable.

```
UPDATE Orders SET OpenDate='1995-01-15' WHERE OrderID=1023
UPDATE Orders SET OpenDate='15-Jan-1995' WHERE OrderID=1023
```

The escape sequence for a date, time, or timestamp literal also can be placed in a character variable bound to a date, time, or timestamp parameter. For
example, the following code uses a date parameter bound to a character variable to update the open date of sales order 1023 in the Orders table:

```sql
SELECT {fn UCASE(Name)} FROM Customers
SELECT uppercase(Name) FROM Customers
```

An application can mix calls to scalar functions that use native syntax and calls to scalar functions that use ODBC syntax. For example, assume that names in the Employee table are stored as a last name, a comma, and a first name. The following SQL statement creates a result set of last names of employees in the Employee table. The statement uses the ODBC scalar function `SUBSTRING` and the SQL Server scalar function `CHARINDEX` and will execute correctly only on SQL Server.
SELECT {fn SUBSTRING(Name, 1, CHARINDEX(',', Name) – 1)} FROM Customers
For maximum interoperability, applications should use the CONVERT scalar function to make sure that the output of a scalar function is the required type. The
CONVERT function converts data from one SQL data type to the specified SQL data type. The syntax of the CONVERT function is
CONVERT(

value_exp

, data_type)

where value_exp is a column name, the result of another scalar function, or a literal value, and data_type is a keyword that matches the #define name that
is used by an SQL data type identifier as defined in Appendix D: Data Types. For example, the following SQL statement uses the CONVERT function to make
sure that the output of the CURDATE function is a date, instead of a timestamp or character data:

INSERT INTO Orders (OrderID, CustID, OpenDate, SalesPerson, Status)
VALUES (?, ?, {fn CONVERT({fn CURDATE()}, SQL_DATE)}, ?, ?)
To determine which scalar functions are supported by a data source, an application calls SQLGetInfo with the SQL_CONVERT_FUNCTIONS,
SQL_NUMERIC_FUNCTIONS, SQL_STRING_FUNCTIONS, SQL_SYSTEM_FUNCTIONS, and SQL_TIMEDATE_FUNCTIONS options. To determine which conversion
operations are supported by the CONVERT function, an application calls SQLGetInfo with any of the options that start with SQL_CONVERT.
© 2016 Microsoft

LIKE Predicate Escape Character
In a LIKE predicate, the percent sign (%) matches zero or more of any character and the underscore (_) matches any one character. To match an actual
percent sign or underscore in a LIKE predicate, an escape character must come before the percent sign or underscore. The escape sequence that defines the
LIKE predicate escape character is:
{escape '

escape-character

'}

where escape-character is any character supported by the data source.
For more information about the LIKE escape sequence, see LIKE Escape Sequence in Appendix C: SQL Grammar.
For example, the following SQL statements create the same result set of customer names that start with the characters "%AAA". The first statement uses the
escape-sequence syntax. The second statement uses the native syntax for Microsoft® Access and is not interoperable. Notice that the second percent
character in each LIKE predicate is a wildcard character that matches zero or more of any character.

SELECT Name FROM Customers WHERE Name LIKE '\%AAA%' {escape '\'}
SELECT Name FROM Customers WHERE Name LIKE '[%]AAA%'
To determine whether the LIKE predicate escape character is supported by a data source, an application calls SQLGetInfo with the
SQL_LIKE_ESCAPE_CLAUSE option.
© 2016 Microsoft

Outer Joins
ODBC supports the SQL-92 left, right, and full outer join syntax. The escape sequence for outer joins is
{oj outer-join}
where outer-join is
table-reference {LEFT | RIGHT | FULL} OUTER JOIN

{table-reference | outer-join} ON search-condition

table-reference specifies a table name, and search-condition specifies the join condition between the table-references.
An outer join request must appear after the FROM keyword and before the WHERE clause (if one exists). For complete syntax information, see Outer Join
Escape Sequence in Appendix C: SQL Grammar.
For example, the following SQL statements create the same result set that lists all customers and shows which has open orders. The first statement uses the
escape-sequence syntax. The second statement uses the native syntax for Oracle and is not interoperable.

SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status
FROM {oj Customers LEFT OUTER JOIN Orders ON Customers.CustID=Orders.CustID}
WHERE Orders.Status='OPEN'
SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status
FROM Customers, Orders


To determine the types of outer joins that a data source and driver support, an application calls `SQLGetInfo` with the SQL_OJ_CAPABILITIES flag. The types of outer joins that might be supported are left, right, full, or nested outer joins; outer joins in which the column names in the `ON` clause do not have the same order as their respective table names in the `OUTER JOIN` clause; inner joins in conjunction with outer joins; and outer joins using any ODBC comparison operator. If the SQL_OJ_CAPABILITIES information type returns 0, no outer join clause is supported.

### Procedure Calls

A procedure is an executable object stored on the data source. Generally, it is one or more SQL statements that have been precompiled. The 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.

For more information about the procedure call escape sequence, see [Procedure Call Escape Sequence](#) in Appendix C: SQL Grammar.

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 `parameter` is an input or an input/output parameter, it can be a literal or a parameter marker. However, interoperable applications should always use parameter markers because some data sources do not accept literal parameter values. If `parameter` is an output parameter, it must be a parameter marker. Parameter markers must be bound with `SQLBindParameter` before the procedure call statement is executed.

Input and output parameters can be omitted from procedure calls. If a procedure is called with parentheses but without any parameters, such as `{call procedure-name()}`, the driver instructs the data source to use the default value for the first parameter. If the procedure does not have any parameters, this might cause the procedure to fail. If a procedure is called without parentheses, such as `{call procedure-name}`, the driver does not send any parameter values.

Literals can be specified for input and input/output parameters in procedure calls. For example, suppose 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:

```
{call InsertOrder(, 10, ?, ?, ?)}  // Not interoperable!
```

Notice 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. Another way to specify the default value of an input or input/output parameter is to set the value of the length/indicator buffer bound to the parameter to SQL_DEFAULT_PARAM.

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.

Suppose the procedure `PARTS_IN_ORDERS` creates a result set that contains a list of orders that contain a particular part number. The following code calls this procedure for part number 544:

```c
SQUINTEGER PartID;
SQUINTEGER PartIDInd = 0;
// Bind the parameter.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0, 0,
                 &PartID, 0, PartIDInd);
// Place the department number in PartID.
PartID = 544;
// Execute the statement.
SQLExecDirect(hstmt, "{call PARTS_IN_ORDERS(?)}", SQL_NTS);
```

To determine whether a data source supports procedures, an application calls `SQLGetInfo` with the SQL_PROCEDURES option.

For more information about procedures, see [Procedures](#).

### Executing Statements ODBC

ODBC applications perform almost all database access by executing SQL statements. The general sequence of events is to allocate a statement handle, set any statement attributes, execute the statement, retrieve any results, and free the statement handle.

This section contains the following topics.

© 2016 Microsoft
Allocating a Statement Handle

Before the application can execute a statement, it must allocate a statement handle as follows:

1. The application declares a variable of type HSTMT. It then calls `SQLAllocHandle` and passes the address of this variable, the handle of the connection in which to allocate the statement, and the SQL_HANDLE_STMT option. For example:

   ```c
   SQLHSTMT hstmt1;
   SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
   ```

2. The Driver Manager allocates a structure in which to store information about the statement and calls `SQLAllocHandle` in the driver with the SQL_HANDLE_STMT option.

3. The driver allocates its own structure in which to store information about the statement and returns the driver statement handle to the Driver Manager.

4. The Driver Manager returns the Driver Manager statement handle to the application in the application variable.

The statement handle identifies which statement to use when calling ODBC functions. For more information about statement handles, see Statement Handles.

Statement Attributes

Statement attributes are characteristics of the statement. For example, whether to use bookmarks and what kind of cursor to use with the statement's result set are statement attributes.

Statement attributes are set with `SQLSetStmtAttr` and their current settings retrieved with `SQLGetStmtAttr`. There is no requirement that an application set any statement attributes; all statement attributes have defaults, some of which are driver-specific.

When a statement attribute can be set depends on the attribute itself. The SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, SQL_ATTR_SIMULATE_CURSOR, and SQL_ATTR_USE_BOOKMARKS statement attributes must be set before the statement is executed. The SQL_ATTR_ASYNC_ENABLE and SQL_ATTR_NOSCAN statement attributes can be set at any time but are not applied until the statement is used again. SQL_ATTR_MAX_LENGTH, SQL_ATTR_MAX_ROWS, and SQL_ATTR_QUERY_TIMEOUT statement attributes can be set at any time, but it is driver-specific whether they are applied before the statement is used again. The remaining statement attributes can be set at any time.

Note

The ability to set statement attributes at the connection level by calling `SQLSetConnectAttr` has been deprecated in ODBC 3.x. ODBC 3.x applications should never set statement attributes at the connection level. ODBC 3.x drivers need only support this functionality if they should work with ODBC 2.x applications. For more information, see SQLSetConnectOption Mapping in Appendix G: Driver Guidelines for Backward Compatibility.

An exception to this is the SQL_ATTR_METADATA_ID and SQL_ATTR_ASYNC_ENABLE attributes, which are both connection attributes and statement attributes and can be set either at the connection level or the statement level.

None of the statement attributes introduced in ODBC 3.x (except for SQL_ATTR_METADATA_ID) can be set at the connection level.

For more information, see the `SQLSetStmtAttr` function description.

Executing a Statement
There are four ways to execute a statement, depending on when they are compiled (prepared) by the database engine and who defines them:

- **Direct Execution** The application defines the SQL statement. It is prepared and executed at run time in a single step.
- **Prepared Execution** The application defines the SQL statement. It is prepared and executed at run time in separate steps. The statement can be prepared once and executed multiple times.
- **Procedures** The application can define and compile one or more SQL statements at development time and store these statements on the data source as a procedure. The procedure is executed one or more times at run time. The application can enumerate available stored procedures using catalog functions.
- **Catalog Functions** The driver writer creates a function that returns a predefined result set. Usually, this function submits a predefined SQL statement or calls a procedure created for this purpose. The function is executed one or more times at run time.

A particular statement (as identified by its statement handle) can be executed any number of times. The statement can be executed with a variety of different SQL statements, or it can be executed repeatedly with the same SQL statement. For example, the following code uses the same statement handle (hstmt1) to retrieve and display the tables in the Sales database. It then reuses this handle to retrieve the columns in a table selected by the user.

```sql
SQLHSTMT hstmt1;
SQLCHAR * Table;

// Create a result set of all tables in the Sales database.
SQLTables(hstmt1, "Sales", SQL_NTS, "sysadmin", SQL_NTS, NULL, 0, NULL, 0);

// Fetch and display the table names; then close the cursor.
// Code not shown.

// Have the user select a particular table.
SelectTable(Table);

// Reuse hstmt1 to create a result set of all columns in Table.
SQLColumns(hstmt1, "Sales", SQL_NTS, "sysadmin", SQL_NTS, Table, SQL_NTS, NULL, 0);

// Fetch and display the column names in Table; then close the cursor.
// Code not shown.
```

And the following code shows how a single handle is used to repeatedly execute the same statement to delete rows from a table.

```sql
SQLHSTMT hstmt1;
SQLUINTEGER OrderID;
SQLINTEGER OrderIDInd = 0;

// Prepare a statement to delete orders from the Orders table.
SQLPrepare(hstmt1, "DELETE FROM Orders WHERE OrderID = ?", SQL_NTS);

// Bind OrderID to the parameter for the OrderID column.
SQLBindParameter(hstmt1, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0, &OrderID, 0, &OrderIDInd);

// Repeatedly execute hstmt1 with different values of OrderID.
while ((OrderID = GetOrderID()) != 0) {
  SQLExecute(hstmt1);
}
```

For many drivers, allocating statements is an expensive task, so reusing the same statement in this manner is usually more efficient than freeing existing statements and allocating new ones. Applications that create result sets on a statement must be careful to close the cursor over the result set before reexecuting the statement; for more information, see Closing the Cursor.

Reusing statements also forces the application to avoid a limitation in some drivers of the number of statements that can be active at one time. The exact definition of “active” is driver-specific, but it often refers to any statement that has been prepared or executed and still has results available. For example, after an INSERT statement has been prepared, it is generally considered to be active; after a SELECT statement has been executed and the cursor is still open, it is generally considered to be active; after a CREATE TABLE statement has been executed, it is not generally considered to be active.

An application determines how many statements can be active on a single connection at one time by calling SQLGetInfo with the SQL_MAX_CONCURRENT_ACTIVITIES option. An application can use more active statements than this limit by opening multiple connections to the data source; because connections can be expensive, however, the effect on performance should be considered.

Applications can limit the amount of time allotted for a statement to execute with the SQL_ATTR_QUERY_TIMEOUT statement attribute. If the timeout period expires before the data source returns the result set, the function executing the SQL statement returns SQLSTATE HYT00 (Timeout expired). By default, there is no timeout.

This section contains the following topics:

- Direct Execution
- Prepared Execution
Direct Execution ODBC

Direct execution is the simplest way to execute a statement. When the statement is submitted for execution, the data source compiles it into an access plan and then executes that access plan.

Direct execution is commonly used by generic applications that build and execute statements at run time. For example, the following code builds an SQL statement and executes it a single time:

```c
SQLCHAR *SQLStatement;
// Build an SQL statement.
BuildStatement(SQLStatement);
// Execute the statement.
SQLExecDirect(hstmt, SQLStatement, SQL_NTS);
```

Direct execution works best for statements that will be executed a single time. Its major drawback is that the SQL statement is parsed every time it is executed. In addition, the application cannot retrieve information about the result set created by the statement (if any) until after the statement is executed; this is possible if the statement is prepared and executed in two separate steps.

To execute a statement directly, the application performs the following actions:

1. Sets the values of any parameters. For more information, see Statement Parameters, later in this section.
2. Calls SQLExecDirect and passes it a string containing the SQL statement.
3. When SQLExecDirect is called, the driver:
   - Modifies the SQL statement to use the data source's SQL grammar without parsing the statement; this includes replacing the escape sequences discussed in Escape Sequences in ODBC. The application can retrieve the modified form of an SQL statement by calling SQLNativeSql. Escape sequences are not replaced if the SQL_ATTR_NOSCAN statement attribute is set.
   - Retrieves the current parameter values and converts them as necessary. For more information, see Statement Parameters, later in this section.
   - Sends the statement and converted parameter values to the data source for execution.
   - Returns any errors. These include sequencing or state diagnostics such as SQLSTATE 24000 (Invalid cursor state), syntactic errors such as SQLSTATE 42000 (Syntax error or access violation), and semantic errors such as SQLSTATE 42S02 (Base table or view not found).

Prepared Execution ODBC

Prepared execution is an efficient way to execute a statement more than once. The statement is first compiled, or prepared, into an access plan. The access plan is then executed one or more times at a later time. For more information about access plans, see Processing an SQL Statement.

Prepared execution is commonly used by vertical and custom applications to repeatedly execute the same, parameterized SQL statement. For example, the following code prepares a statement to update the prices of different parts. It then executes the statement multiple times with different parameter values each time.

```c
SQLREAL Price;
SQLINTEGER PartID;
SQLINTEGER PartIDInd = 0, PriceInd = 0;

// Prepare a statement to update salaries in the Employees table.
SQLPrepare(hstmt, "UPDATE Parts SET Price = ? WHERE PartID = ?", SQL_NTS);

// Bind Price to the parameter for the Price column and PartID to the parameter for the PartID column.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,  // Price, 0, &PriceInd);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 10, 0,  // PartID, 0, &PartIDInd);
```
Prepared execution is faster than direct execution for statements executed more than once, primarily because the statement is compiled only once; statements executed directly are compiled each time they are executed. Prepared execution also can provide a reduction in network traffic because the driver can send an access plan identifier to the data source each time the statement is executed, rather than an entire SQL statement, if the data source supports access plan identifiers.

The application can retrieve the metadata for the result set after the statement is prepared and before it is executed. However, returning metadata for prepared, unexecuted statements is expensive for some drivers and should be avoided by interoperable applications if possible. For more information, see Result Set Metadata.

Prepared execution should not be used for statements executed a single time. For such statements, it is slightly slower than direct execution because it requires an additional ODBC function call.

**Important**

Committing or rolling back a transaction, either by explicitly calling SQLEndTran or by working in auto-commit mode, causes some data sources to delete the access plans for all statements on a connection. For more information, see the SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR options in the SQLGetInfo function description.

To prepare and execute a statement, the application:

1. Calls SQLPrepare and passes it a string containing the SQL statement.
2. Sets the values of any parameters. Parameters can actually be set before or after preparing the statement. For more information, see Statement Parameters, later in this section.
3. Calls SQLExecute and does any additional processing that is necessary, such as fetching data.
4. Repeats steps 2 and 3 as necessary.
5. When SQLPrepare is called, the driver:
   - Modifies the SQL statement to use the data source's SQL grammar without parsing the statement. This includes replacing the escape sequences discussed in Escape Sequences in ODBC. The application can retrieve the modified form of an SQL statement by calling SQLNativeSql. Escape sequences are not replaced if the SQL_ATTR_NOSCAN statement attribute is set.
   - Sends the statement to the data source for preparation.
   - Stores the returned access plan identifier for later execution (if the preparation succeeded) or returns any errors (if the preparation failed). Errors include syntactic errors such as SQLSTATE 42000 (Syntax error or access violation) and semantic errors such as SQLSTATE 42S02 (Base table or view not found).

   **Note**

   Some drivers do not return errors at this point but instead return them when the statement is executed or when catalog functions are called. Thus, SQLPrepare might appear to have succeeded when in fact it has failed.

6. When SQLExecute is called, the driver:
   - Retrieves the current parameter values and converts them as necessary. For more information, see Statement Parameters, later in this section.
   - Sends the access plan identifier and converted parameter values to the data source.
   - Returns any errors. These are generally run-time errors such as SQLSTATE 24000 (Invalid cursor state). However, some drivers return syntactic and semantic errors at this point.

If the data source does not support statement preparation, the driver must emulate it to the extent possible. For example, the driver might do nothing when SQLPrepare is called and then perform direct execution of the statement when SQLExecute is called.

If the data source supports syntax checking without execution, the driver might submit the statement for checking when SQLPrepare is called and submit the statement for execution when SQLExecute is called.

If the driver cannot emulate statement preparation, it stores the statement when SQLPrepare is called and submits it for execution when SQLExecute is called.

Because emulated statement preparation is not perfect, SQLExecute can return any errors normally returned by SQLPrepare.

© 2016 Microsoft

**Procedures ODBC**

A procedure is an executable object stored on the data source. Generally, it is one or more SQL statements that have been precompiled.

This section contains the following topics.
When to Use Procedures

There are a number of advantages to using procedures, all based on the fact that using procedures moves SQL statements from the application to the data source. What is left in the application is an interoperable procedure call. These advantages include:

- **Performance** Procedures are usually the fastest way to execute SQL statements. Like prepared execution, the statement is compiled and executed in two separate steps. Unlike prepared execution, procedures are executed only at run time. They are compiled at a different time.

- **Business Rules** A business rule is a rule about the way in which a company does business. For example, only someone with the title Sales Person might be allowed to add new sales orders. Placing these rules in procedures allows individual companies to customize vertical applications by rewriting the procedures called by the application without having to modify the application code. For example, an order entry application might call the procedure `InsertOrder` with a fixed number of parameters; exactly how `InsertOrder` is implemented can vary from company to company.

- **Replaceability** Closely related to placing business rules in procedures is the fact that procedures can be replaced without recompiling the application. If a business rule changes after a company has bought and installed an application, the company can change the procedure containing that rule. From the application's standpoint, nothing has changed; it still calls a particular procedure to accomplish a particular task.

- **DBMS-specific SQL** Procedures provide a way for applications to exploit DBMS-specific SQL and still remain interoperable. For example, a procedure on a DBMS that supports control-of-flow statements in SQL might trap and recover from errors, while a procedure on a DBMS that does not support control-of-flow statements might simply return an error.

- **Procedures survive transactions** On some data sources, the access plans for all prepared statements on a connection are deleted when a transaction is committed or rolled back. By placing SQL statements in procedures, which are permanently stored in the data source, the statements survive the transaction. Whether the procedures survive in a prepared, partially prepared, or unprepared state is DBMS-specific.

- **Separate development** Procedures can be developed separately from the rest of the application. In large corporations, this might provide a way to further exploit the skills of highly specialized programmers. In other words, application programmers can write user-interface code and database programmers can write procedures.

Procedures are generally used by vertical and custom applications. These applications tend to perform fixed tasks, and it is possible to hard-code procedure calls in them. For example, an order entry application might call the procedures `InsertOrder`, `DeleteOrder`, `UpdateOrder`, and `GetOrders`.

There is little reason to call procedures from generic applications. Procedures are usually written to perform a task in the context of a particular application and so have no use to generic applications. For example, a spreadsheet has no reason to call the `InsertOrder` procedure just mentioned. Furthermore, generic applications should not construct procedures at run time in hopes of providing faster statement execution; not only is this likely to be slower than prepared or direct execution, it also requires DBMS-specific SQL statements.

An exception to this is application-development environments, which often provide a way for programmers to build SQL statements that execute procedures and may provide a way for programmers to test procedures. Such environments call `SQLProcedures` to list available procedures and `SQLProcedureColumns` to list the input, input/output, and output parameters, the procedure return value, and the columns of any result sets created by a procedure. However, such procedures must be developed beforehand on each data source; doing so requires DBMS-specific SQL statements.

There are three major disadvantages to using procedures. The first is that procedures must be written and compiled for each DBMS with which the application is to run. While this is not a problem for custom applications, it can significantly increase development and maintenance time for vertical applications designed to run with a number of DBMSs.

The second disadvantage is that many DBMSs do not support procedures. Again, this is most likely to be a problem for vertical applications designed to run with a number of DBMSs. To determine whether procedures are supported, an application calls `SQLGetInfo` with the `SQL_PROCEDURES` option.

The third disadvantage, which is particularly applicable to application development environments, is that ODBC does not define a standard grammar for creating procedures. That is, although applications can call procedures interoperably, they cannot create them interoperably.

Executing Procedures

ODBC defines a standard escape sequence for executing procedures. For the syntax of this sequence and a code example that uses it, see Procedure Calls.

To execute a procedure, an application performs the following actions:

1. Sets the values of any parameters. For more information, see Statement Parameters, later in this section.

2. Calls `SQLExecDirect` and passes it a string containing the SQL statement that executes the procedure. This statement can use the escape sequence defined by ODBC or DBMS-specific syntax; statements that use DBMS-specific syntax are not interoperable.
3. When SQLExecDirect is called, the driver:

- Retrieves the current parameter values and converts them as necessary. For more information, see Statement Parameters, later in this section.
- Calls the procedure in the data source and sends it the converted parameter values. How the driver calls the procedure is driver-specific. For example, it might modify the SQL statement to use the data source’s SQL grammar and submit this statement for execution, or it might call the procedure directly using a Remote Procedure Call (RPC) mechanism that is defined in the data stream protocol of the DBMS.
- Returns the values of any input/output or output parameters or the procedure return value, assuming the procedure succeeds. These values might not be available until after all other results (row counts and result sets) generated by the procedure have been processed. If the procedure fails, the driver returns any errors.

Batches of SQL Statements

A batch of SQL statements is a group of two or more SQL statements or a single SQL statement that has the same effect as a group of two or more SQL statements. In some implementations, the entire batch statement is executed before any results are available. This is often more efficient than submitting statements separately, because network traffic can often be reduced and the data source can sometimes optimize execution of a batch of SQL statements. In other implementations, calling SQLMoreResults triggers the execution of the next statement in the batch. ODBC supports the following types of batches:

- **Explicit Batches** An explicit batch is two or more SQL statements separated by semicolons (;). For example, the following batch of SQL statements opens a new sales order. This requires inserting rows into both the Orders and Lines tables. Note that there is no semicolon after the last statement.

```sql
INSERT INTO Orders (OrderID, CustID, OpenDate, SalesPerson, Status)
VALUES (2002, 1001, {fn CURDATE()}, 'Garcia', 'OPEN');
INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 1, 1234, 18);
INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 2, 987, 8);
INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 3, 566, 17);
INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 4, 412, 50)
```

- **Procedures** If a procedure contains more than one SQL statement, it is considered to be a batch of SQL statements. For example, the following SQL Server-specific statement creates a procedure that returns a result set containing information about a customer and a result set listing all the open sales orders for that customer:

```sql
CREATE PROCEDURE GetCustInfo (@CustomerID INT)
AS
SELECT * FROM Customers WHERE CustID = @CustomerID
SELECT OrderID FROM Orders
WHERE CustID = @CustomerID AND Status = 'OPEN'
```

The CREATE PROCEDURE statement itself is not a batch of SQL statements. However, the procedure it creates is a batch of SQL statements. No semicolons separate the two SELECT statements because the CREATE PROCEDURE statement is specific to SQL Server, and SQL Server does not require semicolons to separate multiple statements in a CREATE PROCEDURE statement.

- **Arrays of Parameters** Arrays of parameters can be used with a parameterized SQL statement as an effective way to perform bulk operations. For example, arrays of parameters can be used with the following INSERT statement to insert multiple rows into the Lines table while executing only a single SQL statement:

```sql
INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (?, ?, ?, ?)
```

If a data source does not support arrays of parameters, the driver can emulate them by executing the SQL statement once for each set of parameters. For more information, see Statement Parameters and Arrays of Parameter Values, later in this section.

The different types of batches cannot be mixed in an interoperable manner. That is, how an application determines the result of executing an explicit batch that includes procedure calls, an explicit batch that uses arrays of parameters, and a procedure call that uses arrays of parameters is driver-specific.

This section contains the following topics:

- Result-Generating and Result-Free Statements
- Executing Batches
- Errors and Batches
Result-Generating and Result-Free Statements

SQL statements can be loosely divided into the following five categories:

- **Result Set-Generating Statements** These are SQL statements that generate a result set. For example, a `SELECT` statement.
- **Row Count-Generating Statements** These are SQL statements that generate a count of affected rows. For example, an `UPDATE` or `DELETE` statement.
- **Data Definition Language (DDL) Statements** These are SQL statements that modify the structure of the database. For example, `CREATE TABLE` or `DROP INDEX`.
- **Context-Changing Statements** These are SQL statements that change the context of a database. For example, the `USE` and `SET` statements in SQL Server.
- **Administrative Statements** These are SQL statements used for administrative purposes in a database. For example, `GRANT` and `REVOKE`.

SQL statements in the first two categories are collectively known as *result-generating statements*. SQL statements in the latter three categories are collectively known as *result-free statements*. ODBC defines the semantics of batches that include only result-generating statements. These semantics vary widely across data sources, so they are data source-specific. For example, the SQL Server driver does not support dropping an object and then referring to or re-creating the same object in the same batch. Therefore, the term *batch* as used in this manual refers only to batches of result-generating statements.

Executing Batches

Before an application executes a batch of statements, it should first check whether they are supported. To do this, the application calls `SQLGetInfo` with the `SQL_BATCHUPPORT`, `SQL_PARAM_ARRAY_ROW_COUNTS`, and `SQL_PARAM_ARRAY_SELECTS` options. The first option returns whether row count–generating and result set–generating statements are supported in explicit batches and procedures, while the latter two options return information about the availability of row counts and result sets in parameterized execution.

Batches of statements are executed through `SQLExecute` or `SQLExecDirect`. For example, the following call executes an explicit batch of statements to open a new sales order.

```sql
SQLCHAR *BatchStmt =
"INSERT INTO Orders (OrderID, CustID, OpenDate, SalesPerson, Status)
VALUES (2002, 1001, (fn CURDATE()), 'Garcia', 'OPEN');"
"INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 1, 1234, 10);
"INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 2, 987, 8);
"INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 3, 566, 17);
"INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 4, 412, 500);"
SQLExecDirect(hstmt, BatchStmt, SQL_NTS);
```

When a batch of result-generating statements is executed, it returns one or more row counts or result sets. For information about how to retrieve these, see Multiple Results.

If a batch of statements includes parameter markers, these are numbered in increasing parameter order as they are in any other statement. For example, the following batch of statements has parameters numbered from 1 through 21; those in the first `INSERT` statement are numbered 1 through 5 and those in the last `INSERT` statement are numbered 18 through 21.

```sql
INSERT INTO Orders (OrderID, CustID, OpenDate, SalesPerson, Status)
VALUES (?, ?, ?, ?, ?);
INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (?, ?, ?, ?);
INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (?, ?, ?, ?);
INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (?, ?, ?, ?);
```

For more information about parameters, see Statement Parameters, later in this section.

Errors and Batches

When an error occurs while executing a batch of SQL statements, one of the following four outcomes is possible. (Each possible outcome is data source-specific and might even depend on the statements included in the batch.)
- No statements in the batch are executed.
- No statements in the batch are executed and the transaction is rolled back.
- All of the statements before the error statement are executed.
- All of the statements except the error statement are executed.

In the first two cases, `SQLExecute` and `SQLExecuteDirect` return SQL_ERROR. In the latter two cases, they may return SQL_SUCCESS_WITH_INFO or SQL_SUCCESS, depending on the implementation. In all cases, further error information can be retrieved with `SQLGetDiagField`, `SQLGetDiagRec`, or `SQLERROR`. However, the nature and depth of this information is data source-specific. Furthermore, this information is unlikely to exactly identify the statement in error.

© 2016 Microsoft

### Executing Catalog Functions

Because a catalog function creates a result set, it is equivalent to executing any result set–generating SQL statement. In fact, catalog functions are often implemented by executing predefined SQL statements or calling predefined procedures that are shipped with the driver or DBMS. Almost anything that applies to SQL statements that create result sets also applies to catalog functions. For example, the SQL_ATTR_MAX_ROWS statement attribute limits the number of rows returned by the catalog function, just as it limits the number of rows returned by a `SELECT` statement.

To execute a catalog function, an application just calls the function.

For more information about catalog functions, see [Catalog Functions](#).

© 2016 Microsoft

### Statement Parameters

A parameter is a variable in an SQL statement. For example, suppose a Parts table has columns named PartID, Description, and Price. To add a part without parameters would require constructing an SQL statement such as:

```
INSERT INTO Parts (PartID, Description, Price) VALUES (2100, 'Drive shaft', 50.00)
```

Although this statement inserts a new order, it is not a good solution for an order entry application because the values to insert cannot be hard-coded in the application. An alternative is to construct the SQL statement at run time, using the values to be inserted. This is also not a good solution, because of the complexity of constructing statements at run time. The best solution is to replace the elements of the `VALUES` clause with question marks (?), or parameter markers:

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

The parameter markers are then bound to application variables. To add 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.

The statement just shown might be hard-coded in an order entry application to insert a new row. However, parameter markers are not limited to vertical applications. For any application, they ease the difficulty of constructing SQL statements at run time by avoiding conversions to and from text. For example, the part ID just shown is most likely stored in the application as an integer. If the SQL statement is constructed without parameter markers, the application must convert the part ID to text and the data source must convert it back to an integer. By using a parameter marker, the application can send the part ID to the driver as an integer, which usually can send it to the data source as an integer. This saves two conversions. For long data values, this is very important, because the text forms of such values frequently exceed the allowed length of an SQL statement.

Parameters are valid only in certain places in SQL statements. For example, they are not allowed in the select list (the list of columns to be returned by a `SELECT` statement), nor are they allowed as both operands of a binary operator such as the equal sign (=), because it would be impossible to determine the parameter type. Generally, parameters are valid only in Data Manipulation Language (DML) statements, and not in Data Definition Language (DDL) statements. For more information, see [Parameter Markers](#) in Appendix C: SQL Grammar.

When the SQL statement invokes a procedure, named parameters can be used. Named parameters are identified by their names, not by their position in the SQL statement. They can be bound by a call to `SQLBindParameter`, but the parameter is identified by the SQL_DESC_NAME field of the IPD (implementation parameter descriptor), not by the `ParameterNumber` argument of `SQLBindParameter`. They can also be bound by calling `SQLSetDescField` or `SQLSetDescRec`. For more information about named parameters, see [Binding Parameters by Name (Named Parameters)](#), later in this section. For more information about descriptors, see [Descriptors](#).

This section contains the following topics.
Binding Parameters ODBC

Each parameter in an SQL statement must be associated, or bound, to a variable in the application before the statement is executed. When the application binds a variable to a parameter, it describes that variable — address, C data type, and so on — to the driver. It also describes the parameter itself — SQL data type, precision, and so on. The driver stores this information in the structure it maintains for that statement and uses the information to retrieve the value from the variable when the statement is executed.

Parameters can be bound or rebound at any time before a statement is executed. If a parameter is rebound after a statement is executed, the binding does not apply until the statement is executed again. To bind a parameter to a different variable, an application simply rebinds the parameter with the new variable; the previous binding is automatically released.

A variable remains bound to a parameter until a different variable is bound to the parameter, until all parameters are unbound by calling SQLFreeStmt with the SQL_RESET_PARAMS option, or until the statement is released. For this reason, the application must be sure that variables are not freed until after they are unbound. For more information, see Allocating and Freeing Buffers.

Because parameter bindings are just information stored in the structure maintained by the driver for the statement, they can be set in any order. They are also independent of the SQL statement that is executed. For example, suppose an application binds three parameters and then executes the following SQL statement:

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

If the application then immediately executes the SQL statement

```
SELECT * FROM Orders WHERE OrderID = ?, OpenDate = ?, Status = ?
```

on the same statement handle, the parameter bindings for the INSERT statement are used because those are the bindings stored in the statement structure. In most cases, this is a poor programming practice and should be avoided. Instead, the application should call SQLFreeStmt with the SQL_RESET_PARAMS option to unbind all the old parameters and then bind new ones.

This section contains the following topics.

- Binding Parameter Markers
- Binding Parameters by Name (Named Parameters)
- Parameter Binding Offsets
- Describing Parameters

Binding Parameter Markers

The application binds parameters by calling SQLBindParameter. SQLBindParameter binds one parameter at a time. With it, the application specifies the following:

- The parameter number. Parameters are numbered in increasing parameter order in the SQL statement, starting with the number 1. While it is legal to specify a parameter number that is higher than the number of parameters in the SQL statement, the parameter value will be ignored when the statement is executed.
- The parameter type (input, input/output, or output). Except for parameters in procedure calls, all parameters are input parameters. For more information, see Procedure Parameters, later in this section.
- The C data type, address, and byte length of the variable bound to the parameter. The driver must be able to convert the data from the C data type to the SQL data type or an error is returned. For a list of supported conversions, see Converting Data from C to SQL Data Types in Appendix D: Data Types.
The SQL data type, precision, and scale of the parameter itself.

The address of a length/indicator buffer. It provides the byte length of binary or character data, specifies that the data is NULL, or specifies that the data will be sent with SQLPutData. For more information, see Using Length/Indicator Values.

For example, the following code binds SalesPerson and CustID to parameters for the SalesPerson and CustID columns. Because SalesPerson contains character data, which is variable length, the code specifies the byte length of SalesPerson (11) and binds SalesPersonLenOrInd to contain the byte length of the data in SalesPerson. This information is not necessary for CustID because it contains integer data, which is of fixed length.

```sql
CREATE PROCEDURE test @title_id int = 1, @quote char(30) AS <blah>
```

When SQLBindParameter is called, the driver stores this information in the structure for the statement. When the statement is executed, it uses the information to retrieve the parameter data and send it to the data source.

**Note**

In ODBC 1.0, parameters were bound with SQLSetParam. The Driver Manager maps calls between SQLSetParam and SQLBindParameter, depending on the versions of ODBC used by the application and driver.

© 2016 Microsoft

### Binding Parameters by Name (Named Parameters)

Certain DBMSs allow an application to specify the parameters to a stored procedure by name instead of by position in the procedure call. Such parameters are called named parameters. ODBC supports the use of named parameters. In ODBC, named parameters are used only in calls to stored procedures and cannot be used in other SQL statements.

The driver checks the value of the SQL_DESC_UNNAMED field of the IPD to determine whether named parameters are used. If SQL_DESC_UNNAMED is not set to SQL_UNNAMED, the driver uses the name in the SQL_DESC_NAME field of the IPD to identify the parameter. To bind the parameter, an application can call SQLBindParameter to specify the parameter information and then can call SQLSetDescField to set the SQL_DESC_NAME field of the IPD. When named parameters are used, the order of the parameter in the procedure call is not important and the parameter's record number is ignored.

The difference between unnamed parameters and named parameters is in the relationship between the record number of the descriptor and the parameter number in the procedure. When unnamed parameters are used, the first parameter marker is related to the first record in the parameter descriptor, which in turn is related to the first parameter (in creation order) in the procedure call. When named parameters are used, the first parameter marker is still related to the first record of the parameter descriptor, but the relationship between the record number of the descriptor and the parameter number in the procedure does not exist anymore. Named parameters do not use the mapping of the descriptor record number to the procedure parameter position; instead, the descriptor record name is mapped to the procedure parameter name.

**Note**

If automatic population of the IPD is enabled, the driver will populate the descriptor such that the order of the descriptor records will match the order of the parameters in the procedure definition, even if named parameters are used.

If a named parameter is used, all parameters must be named parameters. If any parameter is not a named parameter, then none of the parameters can be named parameters. If there were a mixture of named parameters and unnamed parameters, the behavior would be driver-dependent.

As an example of named parameters, suppose a SQL Server stored procedure has been defined as follows:

```sql
CREATE PROCEDURE test @title_id int = 1, @quote char(30) AS <blah>
```

In this procedure, the first parameter, @title_id, has a default value of 1. An application can use the following code to invoke this procedure such that it specifies only one dynamic parameter. This parameter is a named parameter with the name "@quote".
Parameter Binding Offsets

An application can specify that an offset is added to bound parameter buffer addresses and the corresponding length/indicator buffer addresses when `SQLExecDirect` or `SQLExecute` is called. The result of these additions determines the addresses used in these operations.

Bind offsets allow an application to change bindings without calling `SQLBindParameter` for previously bound parameters. A call to `SQLBindParameter` to rebind a parameter changes the buffer address and the length/indicator pointer. Rebinding with an offset, on the other hand, simply adds an offset to the existing bound parameter buffer address and length/indicator buffer address. When offsets are used, the bindings are a “template” of how the application buffers are laid out and the application can move this “template” to different areas of memory by changing the offset. A new offset can be specified at any time and is always added to the originally bound values.

To specify a bind offset, the application sets the SQL_ATTR_PARAM_BIND_OFFSET_PTR statement attribute to the address of an SQLINTEGER buffer. Before the application calls a function that uses the bindings, it places an offset in bytes in this buffer, as long as neither the parameter buffer address nor the length/indicator buffer address is 0, and the bound parameter is in the SQL statement. The sum of the address and the offset must be a valid address. (This means that either or both the offset and the address to which the offset is added can be invalid, as long as their sum is a valid address.)

**Note**

Binding offsets are not supported by ODBC 2.x drivers.

Describing Parameters

`SQLBindParameter` has arguments that describe the parameter: its SQL type, precision, and scale. The driver uses this information, or metadata, to convert the parameter value to the type needed by the data source. At first glance, it might seem that the driver is in a better position to know the parameter metadata than the application; after all, the driver can easily discover the metadata for a result set column. As it turns out, this is not the case. First, most data sources do not provide a way for the driver to discover parameter metadata. Second, most applications already know the metadata.

If an SQL statement is hard-coded in the application, the application writer already knows the type of each parameter. If an SQL statement is constructed by the application at run time, the application can determine the metadata as it builds the statement. For example, when the application constructs the clause

```sql
WHERE OrderID = ?
```

it can call `SQLColumns` for the OrderID column.

The only situation in which the application cannot easily determine the parameter metadata is when the user enters a parameterized statement. In this case, the application calls `SQLPrepare` to prepare the statement, `SQLNumParams` to determine the number of parameters, and `SQLDescribeParam` to describe each parameter. However, as was noted earlier, most data sources do not provide a way for the driver to discover parameter metadata, so `SQLDescribeParam` is not widely supported.

Setting Parameter Values

To set the value of a parameter, the application simply sets the value of the variable bound to the parameter. It is not important when this value is set, as long as it is set before the statement is executed. The application can set the value before or after binding the variable, and it can change the value as many times as it wants. When the statement is executed, the driver simply retrieves the current value of the variable. This is particularly useful when a prepared statement is executed more than once; the application sets new values for some or all of the variables each time the statement is executed. For an example of this, see Prepared Execution, earlier in this section.
If a length/indicator buffer was bound in the call to `SQLBindParameter`, it must be set to one of the following values before the statement is executed:

- The byte length of the data in the bound variable. The driver checks this length only if the variable is character or binary (`ValueType` is `SQL_C_CHAR` or `SQL_C_BINARY`).
- `SQL_NTS`. The data is a null-terminated string.
- `SQL_NULL_DATA`. The data value is NULL, and the driver ignores the value of the bound variable.
- `SQL_DATA_AT_EXEC` or the result of the `SQL_LEN_DATA_AT_EXEC` macro. The value of the parameter is to be sent with `SQLPutData`. For more information, see Sending Long Data, later in this section.

The following table shows the values of the bound variable and the length/indicator buffer that the application sets for a variety of parameter values.

<table>
<thead>
<tr>
<th>Parameter value</th>
<th>Parameter (SQL) data type</th>
<th>Variable (C) data type</th>
<th>Value in bound variable</th>
<th>Value in length/indicator buffer[d]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ABC&quot;</td>
<td>SQL_CHAR</td>
<td>SQL_C_CHAR</td>
<td>ABC\0[a]</td>
<td>SQL_NTS or 3</td>
</tr>
<tr>
<td>10</td>
<td>SQL_INTEGER</td>
<td>SQL_C_SLONG</td>
<td>10</td>
<td>--</td>
</tr>
<tr>
<td>10</td>
<td>SQL_INTEGER</td>
<td>SQL_C_CHAR</td>
<td>10\0[a]</td>
<td>SQL_NTS or 2</td>
</tr>
<tr>
<td>1 P.M.</td>
<td>SQL_TYPE_TIME</td>
<td>SQL_C_TYPE_TIME</td>
<td>13,0,0[b]</td>
<td>--</td>
</tr>
<tr>
<td>1 P.M.</td>
<td>SQL_TYPE_TIME</td>
<td>SQL_CHAR</td>
<td>{t '13:00:00'}\0[a], [c]</td>
<td>SQL_NTS or 14</td>
</tr>
<tr>
<td>NULL</td>
<td>SQL_SMALLINT</td>
<td>SQL_C_SSHORT</td>
<td>--</td>
<td>SQL_NULL_DATA</td>
</tr>
</tbody>
</table>

[a] "\0" represents a null-termination character. The null-termination character is required only if the value in the length/indicator buffer is `SQL_NTS`.

[b] The numbers in this list are the numbers stored in the fields of the `TIME_STRUCT` structure.

[c] The string uses the ODBC date escape clause. For more information, see Date, Time, and Timestamp Literals.

[d] Drivers must always check this value to see whether it is a special value, such as `SQL_NULL_DATA`.

What a driver does with a parameter value at execution time is driver-dependent. If necessary, the driver converts the value from the C data type and byte length of the bound variable to the SQL data type, precision, and scale of the parameter. In most cases, the driver then sends the value to the data source. In some cases, it formats the value as text and inserts it into the SQL statement before sending the statement to the data source.

© 2016 Microsoft

Sending Long Data

DBMSs define long data as any character or binary data over a certain size, such as 254 characters. It might not be possible to store an entire item of long data in memory, such as when the item represents a long text document or a bitmap. Because such data cannot be stored in a single buffer, the data source sends it to the driver in parts with `SQLPutData` when the statement is executed. Parameters for which data is sent at execution time are known as data-at-execution parameters.

**Note**

An application can actually send any type of data at execution time with `SQLPutData`, although only character and binary data can be sent in parts. However, if the data is small enough to fit in a single buffer, there is generally no reason to use `SQLPutData`. It is much easier to bind the buffer and let the driver retrieve the data from the buffer.

To send data at execution time, the application performs the following actions:

1. Passes a 32-bit value that identifies the parameter in the `ParameterValuePtr` argument in `SQLBindParameter` rather than passing the address of a buffer. This value is not analyzed by the driver; it will be returned to the application later, so it should mean something to the application. For example, it might be the number of the parameter or the handle of a file containing data.

2. Passes the address of a length/indicator buffer in the `StrLen_or_IndPtr` argument of `SQLBindParameter`.

3. Stores `SQL_DATA_AT_EXEC` or the result of the `SQL_LEN_DATA_AT_EXEC(length)` macro in the length/indicator buffer. Both of these values indicate to the driver that the data for the parameter will be sent with `SQLPutData`. `SQL_LEN_DATA_AT_EXEC(length)` is used when sending long data to a data source that needs to know how many bytes of long data will be sent so that it can preallocate space. To determine if a data source requires this value, the application calls `SQLGetInfo` with the `SQL_NEED_LONG_DATA_LEN` option. All drivers must support this macro; if the data source does not require the byte length, the driver can ignore it.

4. Calls `SQLExecute` or `SQLExecDirect`. The driver discovers that a length/indicator buffer contains the value `SQL_DATA_AT_EXEC` or the result of the `SQL_LEN_DATA_AT_EXEC(length)` macro and returns `SQL_NEED_DATA` as the return value of the function.
Calls SQLParamData in response to the SQL_NEED_DATA return value. If long data needs to be sent, SQLParamData returns SQL_NEED_DATA. In the buffer pointed to by the ValuePtr argument, the driver returns the value that identifies the data-at-execution parameter. If there is more than one data-at-execution parameter, the application must use this value to determine which parameter to send data to; the driver is not required to request data for data-at-execution parameters in any particular order.

Calls SQLPutData to send the parameter data to the driver. If the parameter data does not fit into a single buffer, as is often the case with long data, the application calls SQLPutData repeatedly to send the data in parts; it is up to the driver and data source to reassemble the data. If the application passes null-terminated string data, the driver or data source must remove the null-termination character as part of the reassembly process.

Calls SQLParamData again to indicate that it has sent all of the data for the parameter. If there are any data-at-execution parameters for which data has not been sent, the driver returns SQL_NEED_DATA and the value that identifies the next parameter; the application returns to step 6. If data has been sent for all data-at-execution parameters, the statement is executed. SQLParamData returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO and can return any return value or diagnostic that SQLExecute or SQLExecDirect can return.

After SQLExecute or SQLExecDirect returns SQL_NEED_DATA and before data has been completely sent for the last data-at-execution parameter, the statement is in a Need Data state. While a statement is in a Need Data state, the application can call only SQLPutData, SQLParamData, SQLCancel, SQLGetDiagField, or SQLGetDiagRec; all other functions return SQLSTATE HY010 (Function sequence error). Calling SQLCancel cancels execution of the statement and returns it to its previous state. For more information, see Appendix B: ODBC State Transition Tables.

For an example of sending data at execution time, see the SQLPutData function description.

© 2016 Microsoft

Retrieving Output Parameters Using SQLGetData

Before ODBC 3.8, an application could only retrieve the output parameters of a query with a bound output buffer. However, it is difficult to allocate a very large buffer when the size of the parameter value is very large (for example, a large image). ODBC 3.8 introduces a new way to retrieve output parameters in parts. An application can now call SQLGetData with a small buffer multiple times to retrieve a large parameter value. This is similar to retrieving large column data.

To bind an output parameter or input/output parameter to be retrieved in parts, call SQLBindParameter with the InputOutputType argument set to SQL_PARAM_OUTPUT_STREAM or SQL_PARAM_INPUT_OUTPUT_STREAM. With SQL_PARAM_INPUT_OUTPUT_STREAM, an application can use SQLPutData to input data into the parameter, and then use SQLGetData to retrieve the output parameter. The input data must be in the data-at-execution (DAE) form, using SQLPutData instead of binding it to a preallocated buffer.

This feature can be used by ODBC 3.8 applications or recompiled ODBC 3.x and ODBC 2.x applications, and these applications must have an ODBC 3.8 driver that supports retrieving output parameters using SQLGetData and ODBC 3.8 Driver Manager. For information about how to enable an older application to use new ODBC features, see Compatibility Matrix.

Usage Example

For example, consider executing a stored procedure, {CALL sp_f((?,?))}, where both parameters are bound as SQL_PARAM_OUTPUT_STREAM, and the stored procedure returns no result set (later in this topic you will find a more complex scenario):

1. For each parameter, call SQLBindParameter with InputOutputType set to SQL_PARAM_OUTPUT_STREAM and ParameterValuePtr set to a token, such as a parameter number, a pointer to data, or a pointer to a structure that the application uses to bind input parameters. This example will use the parameter ordinal as the token.

2. Execute the query with SQLExecDirect or SQLExecute. SQL_PARAM_DATA_AVAILABLE will be returned, indicating that there are streamed output parameters available for retrieval.

3. Call SQLParamData to get the parameter that is available for retrieval. SQLParamData will return SQL_PARAM_DATA_AVAILABLE with the token of the first available parameter, which is set in SQLBindParameter (step 1). The token is returned in the buffer that the ValuePtr argument points to.

4. Call SQLGetData with the argument ColOrParamNum set to the parameter ordinal to retrieve the data of the first available parameter. If SQLGetData returns SQL_SUCCESS_WITH_INFO and SQLState 01004 (data truncated), and the type is variable length on both the client and server, there is more data to retrieve from the first available parameter. You can continue to call SQLGetData until it returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO with a different SQLState.

5. Repeat step 3 and step 4 to retrieve the current parameter.

6. Call SQLParamData again. If it returns anything except SQL_PARAM_DATA_AVAILABLE, there is no more streamed parameter data to retrieve, and the return code will be the return code of the next statement that is executed.

7. Call SQLMoreResults to process the next set of parameters until it returns SQL_NO_DATA. SQLMoreResults will return SQL_NO_DATA in this example if the statement attribute SQL_ATTR_PARAMSET_SIZE was set to 1. Otherwise, SQLMoreResults will return SQL_PARAM_DATA_AVAILABLE to indicate that there are streamed output parameters available for the next set of parameters to retrieve.

Similar to a DAE input parameter, the token used in the argument ParameterValuePtr in SQLBindParameter (step 1) can be a pointer that points to an application data structure, which contains the ordinal of the parameter and more application-specific information, if necessary.

The order of the returned streamed output or input/output parameters is driver specific and might not always be the same as the order specified in the query.

If the application does not call SQLGetData in step 4, the parameter value is discarded. Similarly, if the application calls SQLParamData before all of a parameter value has been read by SQLGetData, the remainder of the value is discarded, and the application can process the next parameter.

If the application calls SQLMoreResults before all streamed output parameters are processed, SQLParamData does still return
SQL_PARAM_DATA_AVAILABLE), all remaining parameters are discarded. Similarly, if the application calls SQLMoreResults before all of a parameter value has been read by SQLGetData, the remainder of the value and all remaining parameters are discarded, and the application can continue to process the next parameter set.

Note that an application can specify the C data type in both SQLBindParameter and SQLGetData. The C data type specified with SQLGetData overrides the C data type specified in SQLBindParameter, unless the C data type specified in SQLGetData is SQL_APD_TYPE.

Although a streamed output parameter is more useful when the data type of the output parameter is of type BLOB, this functionality can also be used with any data type. The data types supported by streamed output parameters are specified in the driver.

If there are SQL_PARAM_INPUT_OUTPUT_STREAM parameters to be processed, SQLExecute or SQLExecDirect will return SQL_NEED_DATA first. An application can call SQLParamData and SQLPutData to send DAE parameter data. When all DAE input parameters are processed, SQLParamData returns SQL_PARAM_DATA_AVAILABLE to indicate streamed output parameters are available.

When there are streamed output parameters and bound output parameters to be processed, the driver determines the order for processing output parameters. So, if an output parameter is bound to a buffer (the SQLBindParameter parameter InputOutputType is set to SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT), the buffer may not be populated until SQLParamData returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO. An application should read a bound buffer only after SQLParamData returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO that is after all streamed output parameters are processed.

The data source can return a warning and result set, in addition to the streamed output parameter. In general, warnings and result sets are processed separately from a streamed output parameter by calling SQLMoreResults. Process warnings and the result set before processing the streamed output parameter.

The following table describes different scenarios of a single command sent to the server, and how the application should work.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Return value from SQLExecute or SQLExecDirect</th>
<th>What to do next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data only includes streamed output parameters</td>
<td>SQL_PARAM_DATAAVAILABLE</td>
<td>Use SQLParamData and SQLGetData to retrieve streamed output parameters.</td>
</tr>
<tr>
<td>Data includes a result set and streamed output parameters</td>
<td>SQL_SUCCESS</td>
<td>Retrieve the result set with SQLBindCol and SQLGetData.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call SQLMoreResults to start processing streamed output parameters. It should return SQL_PARAM_DATAAVAILABLE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use SQLParamData and SQLGetData to retrieve streamed output parameters.</td>
</tr>
<tr>
<td>Data includes a warning message and streamed output parameters</td>
<td>SQL_SUCCESS_WITH_INFO</td>
<td>Use SQLGetDiagRec and SQLGetDiagField to process warning messages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call SQLMoreResults to start processing streamed output parameters. It should return SQL_PARAM_DATAAVAILABLE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use SQLParamData and SQLGetData to retrieve streamed output parameters.</td>
</tr>
<tr>
<td>Data includes a warning message, result set and streamed output parameters</td>
<td>SQL_SUCCESS_WITH_INFO</td>
<td>Use SQLGetDiagRec and SQLGetDiagField to process warning messages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Retrieve a result set with SQLBindCol and SQLGetData.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call SQLMoreResults to start processing streamed output parameters. SQLMoreResults should return SQL_PARAM_DATAAVAILABLE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use SQLParamData and SQLGetData to retrieve streamed output parameters.</td>
</tr>
<tr>
<td>Query with DAE input parameters, for example, a streamed input/output (DAE) parameter</td>
<td>SQL_NEED_DATA</td>
<td>Call SQLParamData and SQLPutData to send DAE input parameter data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>After all DAE input parameters are processed, SQLParamData can return any return code that SQLExecute and SQLExecDirect can return. The cases in this table can then be applied.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the return code is SQL_PARAM_DATAAVAILABLE, streamed output parameters are available. An application must call SQLParamData again to retrieve the token for the streamed output parameter, as described in the first row of this table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the return code is SQL_SUCCESS, either there is a result set to process or the processing is complete.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the return code is SQL_SUCCESS_WITH_INFO, there are warning messages to process.</td>
</tr>
</tbody>
</table>

After SQLExecute, SQLExecDirect, or SQLMoreResults returns SQL_PARAM_DATAAVAILABLE, a function sequence error will result if an application calls a function that is not in the following list:

- SQLAllocHandle / SQLAllocHandleStd
- SQLDataSources / SQLDrivers
Applications can still use SQLSetDescField or SQLSetDescRec to set the binding information. Field mapping will not be changed. However, fields inside the descriptor might return new values. For example, SQL_DESC_PARAMETER_TYPE might return SQL_PARAM_INPUT_OUTPUT_STREAM or SQL_PARAM_OUTPUT_STREAM.

Usage Scenario: Retrieve an Image in Parts from a Result Set

SQLGetData can be used to get data in parts when a stored procedure returns a result set that contains one row of metadata about an image and the image is returned in a large output parameter.

```c
// CREATE PROCEDURE SP_TestOutputPara
// @ID_of_picture as int,
// @Picture as varbinary(max) out
// AS
// output the image data through streamed output parameter
// GO
BOOL displayPicture(SQLUINTEGER idOfPicture, SQLHSTMT hstmt) {
    SQLLEN lengthOfPicture; // The actual length of the picture.
    BYTE smallBuffer[100]; // A very small buffer.
    SQLRETURN retcode, retcode2;
    // Bind the first parameter (input parameter)
    SQLBindParameter( hstmt, 
        1, // The first parameter.
        SQL_C_ULONG, // The C Data Type.
        SQL_INTEGER, // The SQL Data Type.
        0, // ColumnSize is ignored for integer.
        0, // DecimalDigits is ignored for integer.
        &idOfPicture, // The Address of the buffer for the input parameter.
        0, // BufferLength is ignored for integer.
        NULL); // This is ignored for integer.

    // Bind the streamed output parameter.
    SQLBindParameter( hstmt, 
        2, // The second parameter.
        SQL_PARAM_OUTPUT_STREAM, // A streamed output parameter.
        SQL_C_BINARY, // The C Data Type.
        SQL_VARBINARY, // The SQL Data Type.
        0, // ColumnSize: The maximum size of varbinary(max).
        0, // DecimalDigits is ignored for binary type.
        (SQLPOINTER)2, // ParameterValuePtr: An application-defined token (this will be returned from SQLParamData).
        // In this example, we used the ordinal of the parameter.
        0, // BufferLength is ignored for streamed output parameters.
        &lengthOfPicture); // Strlen_or_Ptr: The status variable returned.

    retcode = SQLPrepare(hstmt, L"(call SP_TestOutputPara(?, ?))", SQL_NTS);
    if ( retcode == SQL_ERROR )
```

```c
/ SQLGetInfo / SQLGetFunctions
/ SQLGetConnectAttr / SQLGetEnvAttr / SQLGetDescField / SQLGetDescRec
/ SQLNumParams
/ SQLDescribeParam
/ SQLNativeSql
/ SQLMoreResults
/ SQLGetDiagField / SQLGetDiagRec
/ SQLCancel
/ SQLCancelHandle (with statement handle)
/ SQLFreeStmt (with Option = SQL_CLOSE, SQL_DROP or SQL_UNBIND)
/ SQLCloseCursor
/ SQLDisconnect
/ SQLFreeHandle (with HandleType = SQL_HANDLE_STMT)
/ SQLGetStmtAttr
```
return FALSE;

retcode = SQLExecute(hstmt);
if ( retcode == SQL_ERROR )
    return FALSE;

// Assume that the retrieved picture exists. Use SQLBindCol or SQLGetData to retrieve the result-set.

// Process the result set and move to the streamed output parameters.
retcode = SQLMoreResults(hstmt);

// SQLGetData retrieves and displays the picture in parts.
// The streamed output parameter is available.
while ( retcode == SQL_PARAM_DATA_AVAILABLE )
{
    SQLPOINTER token; // Output by SQLParamData.
    SQLLEN cbLeft; // #bytes remained
    retcode = SQLParamData(hstmt, &token); // returned token is 2 (according to the binding)
    if ( retcode == SQL_PARAM_DATA_AVAILABLE )
    {
        // A do-while loop retrieves the picture in parts.
        do{
            retcode2 = SQLGetData(hstmt, (UWORD)token, SQL_C_BINARY, smallBuffer, sizeof(smallBuffer), &cbLeft);
        }
        while ( retcode2 == SQL_SUCCESS_WITH_INFO );
    }
}

return TRUE;

Usage Scenario: Send and Receive a Large Object as a Streamed Input/Output Parameter

SQLGetData can be used to get and send data in parts when a stored procedure passes a large object as an input/output parameter, streaming the value to and from the database. You do not have to store all of the data in memory.

// CREATE PROCEDURE SP_TestInOut
// @picture as varbinary(max) out
// AS
// output the image data through output parameter
// go

BOOL displaySimilarPicture(BYTE* image, ULONG lengthOfImage, SQLHSTMT hstmt) {
    BYTE smallBuffer[100]; // A very small buffer.
    SQLRETURN retcode, retcode2;
    SQLRETURN statusOfPicture;

    // First bind the parameters, before preparing the statement that binds the output streamed parameter.
    SQLBindParameter( hstmt,
        1, // The first parameter.
        SQL_C_BINARY, // The C Data Type.
        SQL_VARBINARY, // The SQL Data Type.
        0, // ColumnSize: The maximum size of varbinary(max).
        0, // DecimalDigits is ignored.
        (SQLPOINTER)1, // An application defined token.
        0, // Bufferlength is ignored for streamed I/O parameters.
        &statusOfPicture ); // The status variable.

    statusOfPicture = SQL_DATA_AT_EXEC; // Input data in parts (DAE parameter at input).

    retcode = SQLPrepare(hstmt, L"(call SP_TestInOut(?) )", SQL_NTS);
    if ( retcode == SQL_ERROR )
        return FALSE;

    // Execute the statement.
    retcode = SQLExecute(hstmt);
    if ( retcode == SQL_ERROR )
        return FALSE;

    if ( retcode == SQL_NEED_DATA )
    {
        // Use SQLParamData to loop through DAE input parameters. For each, use SQLPutData to send the data to database in parts.

        // This example uses an I/O parameter with streamed output.
Parameters in procedure calls can be input, input/output, or output parameters. This is different from parameters in all other SQL statements, which are always input parameters.

Input parameters are used to send values to the procedure. For example, suppose the Parts table has PartID, Description, and Price columns. The InsertPart procedure might have an input parameter for each column in the table. For example:

```sql
{call InsertPart(?, ?, ?)}
```

A driver should not modify the contents of an input buffer until `SQLExecDirect` or `SQLExecute` returns `SQL_SUCCESS`, `SQL_SUCCESS_WITH_INFO`, `SQL_ERROR`, `SQL_INVALID_HANDLE`, or `SQL_NO_DATA`. The contents of the input buffer should not be modified while `SQLExecDirect` or `SQLExecute` returns `SQL_NEED_DATA` or `SQL_STILL_EXECUTING`.

Input/output parameters are used both to send values to procedures and retrieve values from procedures. Using the same parameter as both an input and an output parameter tends to be confusing and should be avoided. For example, suppose a procedure accepts an order ID and returns the ID of the customer. This can be defined with a single input/output parameter:

```sql
{call GetCustID(?)}
```

It might be better to use two parameters: an input parameter for the order ID and an output or input/output parameter for the customer ID:

```sql
{call GetCustID(?, ?)}
```

Output parameters are used to retrieve the procedure return value and to retrieve values from procedure arguments; procedures that return values are sometimes known as functions. For example, suppose the `GetCustID` procedure just mentioned returns a value that indicates whether it was able to find the order. In the following call, the first parameter is an output parameter used to retrieve the procedure return value, the second parameter is an input parameter used to specify the order ID, and the third parameter is an output parameter used to retrieve the customer ID:

```sql
{? = call GetCustID(?)}
```
Drivers handle values for input and input/output parameters in procedures no differently than input parameters in other SQL statements. When the statement is executed, they retrieve the values of the variables bound to these parameters and send them to the data source.

After the statement has been executed, drivers store the returned values of input/output and output parameters in the variables bound to those parameters. These returned values are not guaranteed to be set until after all results returned by the procedure have been fetched and SQLMoreResults has returned SQL_NO_DATA. If executing the statement results in an error, the contents of the input/output parameter buffer or output parameter buffer are undefined.

An application calls SQLProcedure to determine whether a procedure has a return value. It calls SQLProcedureColumns to determine the type (return value, input, input/output, or output) of each procedure parameter.

Arrays of Parameter Values

It is often useful for applications to pass arrays of parameters. For example, using arrays of parameters and a parameterized INSERT statement, an application can insert a number of rows at once. There are several advantages to using arrays. First, network traffic is reduced because the data for many statements is sent in a single packet (if the data source supports parameter arrays natively). Second, some data sources can execute SQL statements using arrays faster than executing the same number of separate SQL statements. Finally, when the data is stored in an array, as is often the case for screen data, the application can bind all of the rows in a particular column with a single call to SQLBindParameter and update them by executing a single statement.

Unfortunately, not many data sources support parameter arrays. However, a driver can emulate parameter arrays by executing an SQL statement once for each set of parameter values. This can lead to increases in speed because the driver can then prepare the statement that it plans to execute once for each parameter set. It might also lead to simpler application code.

This section contains the following topics.

- Binding Arrays of Parameters
- Using Arrays of Parameters

Binding Arrays of Parameters

Applications that use arrays of parameters bind the arrays to the parameters in the SQL statement. There are two binding styles:

- Bind an array to each parameter. Each data structure (array) contains all the data for a single parameter. This is called column-wise binding because it binds a column of values for a single parameter.
- Define a structure to hold the parameter data for an entire set of parameters and bind an array of these structures. Each data structure contains the data for a single SQL statement. This is called row-wise binding because it binds a row of parameters.

As when the application binds single variables to parameters, it calls SQLBindParameter to bind arrays to parameters. The only difference is that the addresses passed are array addresses, not single-variable addresses. The application sets the SQL_ATTR_PARAM_BIND_TYPE statement attribute to specify whether it is using column-wise (the default) or row-wise binding. Whether to use column-wise or row-wise binding is largely a matter of application preference. Depending on how the processor accesses memory, row-wise binding might be faster. However, the difference is likely to be negligible except for very large numbers of rows of parameters.

Column-Wise Binding

When using column-wise binding, an application binds one or two arrays to each parameter for which data is to be provided. The first array holds the data values, and the second array holds length/indicator buffers. Each array contains as many elements as there are values for the parameter.

Column-wise binding is the default. The application also can change from row-wise binding to column-wise binding by setting the SQL_ATTR_PARAM_BIND_TYPE statement attribute. The following illustration shows how column-wise binding works.

For example, the following code binds 10-element arrays to parameters for the PartID, Description, and Price columns, and executes a statement to insert 10 rows. It uses column-wise binding.
Row-Wise Binding

When using row-wise binding, an application defines a structure for each set of parameters. The structure contains one or two elements for each parameter. The first element holds the parameter value, and the second element holds the length/indicator buffer. The application then allocates an array of these structures, which contains as many elements as there are values for each parameter.
The application declares the size of the structure to the driver with the SQL_ATTR_PARAM_BIND_TYPE statement attribute. The application binds the addresses of the parameters in the first structure of the array. Thus, the driver can calculate the address of the data for a particular row and column as

\[
\text{Address} = \text{Bound Address} + ((\text{Row Number} - 1) \times \text{Structure Size}) + \text{Offset}
\]

where rows are numbered from 1 to the size of the parameter set. The offset, if defined, is the value pointed to by the SQL_ATTR_PARAM_BIND_OFFSET_PTR statement attribute. The following illustration shows how row-wise binding works. The parameters can be placed in the structure in any order but are shown in sequential order for clarity.

The following code creates a structure with elements for the values to store in the PartID, Description, and Price columns. It then allocates a 10-element array of these structures and binds it to parameters for the PartID, Description, and Price columns, using row-wise binding. It then executes a statement to insert 10 rows.

```c
#define DESC_LEN 51
#define ARRAY_SIZE 10
typedef tagPartStruct {
    SQLREAL Price;
    SQLINTEGER PartID;
    SQLCHAR Desc[DESC_LEN];
    SQLINTEGER PriceInd;
    SQLINTEGER PartIDInd;
    SQLINTEGER DescLenOrInd;
} PartStruct;
PartStruct PartArray[ARRAY_SIZE];
SQLCHAR * Statement = "INSERT INTO Parts (PartID, Description, Price) " "VALUES (?, ?, ?)";
SQLUSMALLINT i, ParamStatusArray[ARRAY_SIZE];
SQLULEN ParamsProcessed;
// Set the SQL_ATTR_PARAM_BIND_TYPE statement attribute to use
// column-wise binding.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_BIND_TYPE, sizeof(PartStruct), 0);
// Specify the number of elements in each parameter array.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, ARRAY_SIZE, 0);
// Specify an array in which to return the status of each set of
// parameters.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_STATUS_PTR, ParamStatusArray, 0);
// Specify an SQLINTEGER value in which to return the number of sets of
// parameters processed.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, &ParamsProcessed, 0);
// Bind the parameters in row-wise fashion.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
    &PartArray[0].PartID, PartID, &PartArray[0].PartIDInd);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, DESC_LEN - 1, 0,
    PartArray[0].Desc, DESC_LEN, &PartArray[0].DescLenOrInd);
SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,
    &PartArray[0].Price, 0, &PartArray[0].PriceInd);
// Set part ID, description, and price.
for (i = 0; i < ARRAY_SIZE; i++) {
    GetNewValues(&PartArray[i].PartID, PartArray[i].Desc, &PartArray[i].Price);
    PartArray[i].PartIDInd = 0;
    PartArray[i].DescLenOrInd = SQL_NTS;
    PartArray[i].PriceInd = 0;
}
// Execute the statement.
SQLExecDirect(hstmt, Statement, SQL_NTS);
```
Using Arrays of Parameters

To use arrays of parameters, the application calls `SQLSetStmtAttr` with an Attribute argument of `SQL_ATTR_PARAMSET_SIZE` to specify the number of sets of parameters. It calls `SQLSetStmtAttr` with an Attribute argument of `SQL_ATTR_PARAMS_PROCESSED_PTR` to specify the address of a variable in which the driver can return the number of sets of parameters processed, including error sets. It calls `SQLSetStmtAttr` with an Attribute argument of `SQL_ATTR_PARAM_STATUS_PTR` to point to an array in which to return status information for each row parameter values. The driver stores these addresses in the structure it maintains for the statement.

**Note**

In ODBC 2.x, `SQLParamOptions` was called to specify multiple values for a parameter. In ODBC 3.x, the call to `SQLParamOptions` has been replaced by calls to `SQLSetStmtAttr` to set the `SQL_ATTR_PARAMSET_SIZE` and `SQL_ATTR_PARAMS_PROCESSED_ARRAY` attributes.

Before executing the statement, the application sets the value of each element of each bound array. When the statement is executed, the driver uses the information it stored to retrieve the parameter values and send them to the data source; if possible, the driver should send these values as arrays. Although the use of arrays of parameters is best implemented by executing the SQL statement with all of the parameters in the array with a single call to the data source, this capability is not widely available in DBMSs today. However, drivers can simulate it by executing an SQL statement multiple times, each with a single set of parameters.

Before an application uses arrays of parameters, it must be sure that they are supported by the drivers used by the application. There are two ways to do this:

- Use only drivers known to support arrays of parameters. The application can hard-code the names of these drivers, or the user can be instructed to use only these drivers. Custom applications and vertical applications commonly use a limited set of drivers.

- Check for support of arrays of parameters at run time. A driver supports arrays of parameters if it is possible to set the `SQL_ATTR_PARAMSET_SIZE` statement attribute to a value greater than 1. Generic applications and vertical applications commonly check for support of arrays of parameters at run time.

The availability of row counts and result sets in parameterized execution can be determined by calling `SQLGetInfo` with the `SQL_PARAM_ARRAY_ROW_COUNTS` and `SQL_PARAM_ARRAY_SELECTS` options. For `INSERT`, `UPDATE`, and `DELETE` statements, the `SQL_PARAM_ARRAY_ROW_COUNTS` option indicates whether individual row counts (one for each parameter set) are available (`SQL_PARC_BATCH`) or whether row counts are rolled up into one (`SQL_PARC_NO_BATCH`). For `SELECT` statements, the `SQL_PARAM_ARRAY_SELECTS` option indicates whether a result set is available for each set of parameters (`SQL_PAS_BATCH`) or whether only one result set is available (`SQL_PAS_NO_BATCH`). If the driver does not allow result set-generating statements to be executed with an array of parameters, `SQL_PARAM_ARRAY_SELECTS` returns `SQL_PAS_NO_SELECT`. It is data source–specific whether arrays of parameters can be used with other types of statements, especially because the use of parameters in these statements would be data source–specific and would not follow ODBC SQL grammar.

The array pointed to by the `SQL_ATTR_PARAM_OPERATION_PTR` statement attribute can be used to ignore rows of parameters. If an element of the array is set to `SQL_PARAM_IGNORE`, the set of parameters corresponding to that element is excluded from the `SQLExecute` or `SQLExecDirect` call. The array pointed to by the `SQL_ATTR_PARAM_OPERATION_PTR` attribute is allocated and filled in by the application and read by the driver. If fetched rows are used as input parameters, the values of the row status array can be used in the parameter operation array.

**Error Processing**

If an error occurs while executing the statement, the execution function returns an error and sets the row number variable to the number of the row containing the error. It is data source–specific whether all rows except the error set are executed or whether all rows before (but not after) the error set are executed. Because it processes sets of parameters, the driver sets the buffer specified by the `SQL_ATTR_PARAMS_PROCESSED_PTR` statement attribute to the number of the row currently being processed. If all sets except the error set are executed, the driver sets this buffer to `SQL_ATTR_PARAMSET_SIZE` after all rows are processed.
If the SQL_ATTR_PARAM_STATUS_PTR statement attribute has been set, SQLExecute or SQLExecDirect returns the parameter status array, which provides the status of each set of parameters. The parameter status array is allocated by the application and filled in by the driver. Its elements indicate whether the SQL statement was executed successfully for the row of parameters or whether an error occurred while processing the set of parameters. If an error occurred, the driver sets the corresponding value in the parameter status array to SQL_PARAM_ERROR and returns SQL_SUCCESS_WITH_INFO. The application can check the status array to determine which rows were processed. Using the row number, the application can often correct the error and resume processing.

How the parameter status array is used is determined by the SQL_PARAM_ARRAY_ROW_COUNTS and SQL_PARAM_ARRAY_SELECTS options returned by a call to SQLGetInfo. For INSERT, UPDATE, and DELETE statements, the parameter status array is filled in with status information if SQL_PARAM_BATCH is returned for SQL_PARAM_ARRAY_ROW_COUNTS, but not if SQL_PARAM_NO_BATCH is returned. For SELECT statements, the parameter status array is filled in if SQL_PARAM_SELECT is returned for SQL_PARAM_ARRAY_SELECT, but not if SQL_PARAM_NO_SELECT or SQL_PARAM_NO_SELECT is returned.

Data-at-Execution Parameters

If any of the values in the length/indicator array are SQL_DATA_AT_EXEC or the result of the SQL_LEN_DATA_AT_EXEC(length) macro, the data for those values is sent with SQLPutData in the usual way. The following aspects of this process bear special comment because they are not readily obvious:

- When the driver returns SQL_NEED_DATA, it must set the address of the row number variable to the row for which it needs data. As in the single-valued case, the application cannot make any assumptions about the order in which the driver will request parameter values within a single set of parameters. If an error occurs in the execution of a data-at-execution parameter, the buffer specified by the SQL_ATTR_PARAMS_PROCESSED_PTR statement attribute is set to the number of the row on which the error occurred, the status for the row in the row status array specified by the SQL_ATTR_PARAM_STATUS_PTR statement attribute is set to SQL_PARAM_ERROR, and the call to SQLExecute, SQLExecDirect, SQLParamData, or SQLPutData returns SQL_ERROR. The contents of this buffer are undefined if SQLExecute, SQLExecDirect, or SQLParamData return SQL_STILL_EXECUTING.

- Because the driver does not interpret the value in the ParameterValuePtr argument of SQLBindParameter for data-at-execution parameters, if the application provides a pointer to an array, SQLParamData does not extract and return an element of this array to the application. Instead, it returns the scalar value the application had supplied. This means the value returned by SQLParamData is not sufficient to specify the parameter for which the application needs to send data; the application also needs to consider the current row number.

When only some of the elements of an array of parameters are data-at-execution parameters, the application must pass the address of an array in ParameterValuePtr that contains elements for all the parameters. This array is interpreted normally for the parameters that are not data-at-execution parameters. For the data-at-execution parameters, the value that SQLParamData provides to the application, which normally could be used to identify the data that the driver is requesting on this occasion, is always the address of the array.

Asynchronous Execution

ODBC supports asynchronous operations for both statement and connection operations. There are two ways to determine when an asynchronous operation is complete:

1. Asynchronous Execution (Polling Method)
2. Asynchronous Execution (Notification Method)

See Also

Executing Statements ODBC

Asynchronous Execution (Polling Method)

Prior to ODBC 3.8 and the Windows 7 SDK, asynchronous operations were permitted only on statement functions. For more information, see the Executing Statement Operations Asynchronously, later in this topic.

ODBC 3.8 in the Windows 7 SDK introduced asynchronous execution on connection-related operations. For more information, see the Executing Connection Operations Asynchronously section, later in this topic.

In the Windows 7 SDK, for asynchronous statement or connection operations, an application determined that the asynchronous operation was complete using the polling method. Beginning in the Windows 8 SDK, you can determine that an asynchronous operation is complete using the notification method. For more information, see Asynchronous Execution (Notification Method).

By default, drivers execute ODBC functions synchronously; that is, the application calls a function and the driver does not return control to the application until it has finished executing the function. However, some functions can be executed asynchronously; that is, the application calls the function, and the driver, after minimal processing, returns control to the application. The application can then call other functions while the first function is still executing.

Asynchronous execution is supported for most functions that are largely executed on the data source, such as the functions to establish connections, prepare
and execute SQL statements, retrieve metadata, fetch data, and commit transactions. It is most useful when the task being executed on the data source takes a long time, such as a login process or a complex query against a large database.

When the application executes a function with a statement or connection that is enabled for asynchronous processing, the driver performs a minimal amount of processing (such as checking arguments for errors), hands processing to the data source, and returns control to the application with the SQL_STILL_EXECUTING return code. The application then performs other tasks. To determine when the asynchronous function has finished, the application polls the driver at regular intervals by calling the function with the same arguments as it originally used. If the function is still executing, it returns SQL_STILL_EXECUTING; if it has finished executing, it returns the code it would have returned had it executed synchronously, such as SQL_SUCCESS, SQL_ERROR, or SQL_NEED_DATA.

Whether a function executes synchronously or asynchronously is driver specific. For example, suppose the result set metadata is cached in the driver. In this case, it takes very little time to execute SQLDescribeCol and the driver should simply execute the function rather than artificially delay execution. On the other hand, if the driver needs to retrieve the metadata from the data source, it should return control to the application while it is doing this. Therefore, the application must be able to handle a return code other than SQL_STILL_EXECUTING when it first executes a function asynchronously.

### Executing Statement Operations Asynchronously

The following statement functions operate on a data source and can execute asynchronously:

<table>
<thead>
<tr>
<th>SQLBulkOperations</th>
<th>SQLColAttribute</th>
<th>SQLColumnPrivileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLColumns</td>
<td>SQLDescribeCol</td>
<td>SQLDescribeParam</td>
</tr>
<tr>
<td>SQLExecDirect</td>
<td>SQLExecute</td>
<td>SQLFetch</td>
</tr>
<tr>
<td>SQLFetchScroll</td>
<td>SQLForeignKeys</td>
<td>SQLGetData</td>
</tr>
<tr>
<td>SQLErrorTypeInfo</td>
<td>SQLMoreResults</td>
<td>SQLNumParams</td>
</tr>
<tr>
<td>SQLNumResultCols</td>
<td>SQLParamData</td>
<td>SQLPrepare</td>
</tr>
<tr>
<td>SQLPrimaryKeys</td>
<td>SQLProcedureColumns</td>
<td>SQLProcedures</td>
</tr>
<tr>
<td>SQLPutData</td>
<td>SQLSetPos</td>
<td>SQLSpecialColumns</td>
</tr>
<tr>
<td>SQLStatistics</td>
<td>SQLTablePrivileges</td>
<td>SQLTables</td>
</tr>
</tbody>
</table>

Asynchronous statement execution is controlled on either a per-statement or a per-connection basis, depending on the data source. That is, the application specifies not that a particular function is to be executed asynchronously, but that any function executed on a particular statement is to be executed asynchronously. To find out which one is supported, an application calls SQLGetInfo with an option of SQL_ASYNC_MODE. SQL_AM_CONNECTION is returned if connection-level asynchronous execution (for a statement handle) is supported; SQL_AM_STATEMENT if statement-level asynchronous execution is supported.

To specify that functions executed with a particular statement are to be executed asynchronously, the application calls SQLSetStmtAttr with the SQL_ATTR_ASYNC_ENABLE attribute and sets it to SQL_ASYNC_ENABLE_ON. If connection-level asynchronous processing is supported, the SQL_ATTR_ASYNC_ENABLE statement attribute is read-only and its value is the same as the connection attribute of the connection on which the statement was allocated. It is driver-specific whether the value of the statement attribute is set at statement allocation time or later. Attempting to set it will return SQL_ERROR and SQLSTATE HYC00 (Optional feature not implemented).

To specify that functions executed with a particular connection are to be executed asynchronously, the application calls SQLSetConnectAttr with the SQL_ATTR_ASYNC_ENABLE attribute and sets it to SQL_ASYNC_ENABLE_ON. All future statement handles allocated on the connection will be enabled for asynchronous execution; it is driver-defined whether existing statement handles will be enabled by this action. If SQL_ATTR_ASYNC_ENABLE is set to SQL_ASYNC_ENABLE_OFF, all statements on the connection are in synchronous mode. An error is returned if asynchronous execution is enabled while there is an active statement on the connection.

To determine the maximum number of active concurrent statements in asynchronous mode that the driver can support on a given connection, the application calls SQLGetInfo with the SQL_MAX_ASYNC_CONCURRENT_STATEMENTS option.

The following code demonstrates how the polling model works:

```sql
SQLHSTMT hstmt1;
SQLRETURN rc;

// Specify that the statement is to be executed asynchronously.
SQLSetStmtAttr(hstmt1, SQL_ATTR_ASYNC_ENABLE, SQL_ASYNC_ENABLE_ON, 0);

// Execute a SELECT statement asynchronously.
while (((rc=SQLExecDirect(hstmt1,"SELECT * FROM Orders",SQL_NTS))==SQL_STILL_EXECUTING) {
    // While the statement is still executing, do something else.
    // Do not use hstmt1, because it is being used asynchronously.
}

// When the statement has finished executing, retrieve the results.
```

While a function is executing asynchronously, the application can call functions on any other statements. The application can also call functions on any
connection, except the one associated with the asynchronous statement. But the application can only call the original function and the following functions (with the statement handle or its associated connection, environment handle), after a statement operation returns SQL_STILL_EXECUTING:

- SQLCancel
- SQLCancelHandle (on the statement handle)
- SQLGetDiagField
- SQLGetDiagRec
- SQLAllocHandle
- SQLGetEnvAttr
- SQLGetConnectAttr
- SQLDataSources
- SQLDrivers
- SQLGetInfo
- SQLGetFunctions
- SQLNativeSql

If the application calls any other function with the asynchronous statement or with the connection associated with that statement, the function returns SQLSTATE HY010 (Function sequence error), for example.

```c
SQLHDBC      hdbc1, hdbc2;
SQLHSTMT     hstmt1, hstmt2, hstmt3;
SQLCHAR     * SQLStatement = "SELECT * FROM Orders";
SQLINTEGER   InfoValue;
SQLRETURN    rc;

SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt2);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc2, &hstmt3);

// Specify that hstmt1 is to be executed asynchronously.
SQLSetStmtAttr(hstmt1, SQL_ATTR_ASYNC_ENABLE, SQL_ASYNC_ENABLE_ON, 0);

// Execute hstmt1 asynchronously.
while ((rc = SQLExecDirect(hstmt1, SQLStatement, SQL_NTS)) == SQL_STILL_EXECUTING) {
    // The following calls return HY010 because the previous call to SQLExecDirect is still executing asynchronously on hstmt1. The first call uses hstmt1 and the second call uses hdbc1, on which hstmt1 is allocated.
    SQLExecDirect(hstmt1, SQLStatement, SQL_NTS);    // Error!
    SQLGetInfo(hdbc1, SQL_UNION, (SQLPOINTER) &InfoValue, 0, NULL);    // Error!
}

// The following calls do not return errors. They use a statement handle other than hstmt1 or a connection handle other than hdbc1.
SQLExecDirect(hstmt2, SQLStatement, SQL_NTS);  // OK
SQLTables(hstmt3, NULL, 0, NULL, 0, NULL, 0, NULL, 0, 0, NULL, 0, NULL, 0);  // OK
SQLGetInfo(hdbc2, SQL_UNION, (SQLPOINTER) &InfoValue, 0, NULL);  // OK
```

When an application calls a function to determine whether it is still executing asynchronously, it must use the original statement handle. This is because asynchronous execution is tracked on a per-statement basis. The application must also supply valid values for the other arguments — the original arguments will do — to get past error checking in the Driver Manager. However, after the driver checks the statement handle and determines that the statement is executing asynchronously, it ignores all other arguments.

While a function is executing asynchronously — that is, after it has returned SQL_STILL_EXECUTING and before it returns a different code — the application can cancel it by calling SQLCancel or SQLCancelHandle with the same statement handle. This is not guaranteed to cancel function execution. For example, the function might have already finished. Furthermore, the code returned by SQLCancel or SQLCancelHandle only indicates whether the attempt to cancel the function was successful, not whether it actually canceled the function. To determine whether the function was canceled, the application calls the function again. If the function was canceled, it returns SQL_ERROR and SQLSTATE HY008 (Operation canceled). If the function was not canceled, it returns another code, such as SQL_SUCCESS, SQL_STILL_EXECUTING, or SQL_ERROR with a different SQLSTATE.

To disable asynchronous execution of a particular statement when the driver supports statement-level asynchronous processing, the application calls SQLSetStmtAttr with the SQL_ATTR_ASYNC_ENABLE attribute and sets it to SQL_ASYNC_ENABLE_OFF. If the driver supports connection-level asynchronous processing, the application calls SQLSetConnectAttr to set SQL_ATTR_ASYNC_ENABLE to SQL_ASYNC_ENABLE_OFF, which disables asynchronous execution of all statements on the connection.

The application should process diagnostic records in the repeating loop of the original function. If SQLGetDiagField or SQLGetDiagRec is called when an asynchronous function is executing, it will return the current list of diagnostic records. Each time the original function call is repeated, it clears previous diagnostic records.
Executing Connection Operations Asynchronously

Before ODBC 3.8, asynchronous execution was allowed for statement-related operations such as prepare, execute, and fetch, as well as for catalog metadata operations. Starting in ODBC 3.8, asynchronous execution is also possible for connection-related operations such as connect, disconnect, commit, and rollback.

For more information on ODBC 3.8, see What’s New in ODBC 3.8.

Executing connection operations asynchronously is useful in the following scenarios:

- When a small number of threads manages a large number of devices with very high data rates. To maximize responsiveness and scalability it is desirable for all operations to be asynchronous.
- When you want to overlap database operations over multiple connections to reduce elapsed transfer times.
- Efficient asynchronous ODBC calls and the ability to cancel connection operations enable an application to allow the user to cancel any slow operation without having to wait for timeouts.

The following functions, which operate on connection handles, can now be executed asynchronously:

<table>
<thead>
<tr>
<th>SQLBrowseConnect</th>
<th>SQLConnect</th>
<th>SQLDisconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLDriverConnect</td>
<td>SQLEndTran</td>
<td>SQLSetConnectAttr</td>
</tr>
</tbody>
</table>

To determine whether a driver supports asynchronous operations on these functions, an application calls SQLGetInfo with SQL_ASYNC_DBC_FUNCTIONS. SQL_ASYNC_DBC_CAPABLE is returned if asynchronous operations are supported. SQL_ASYNC_DBC_NOT_CAPABLE is returned if asynchronous operations are not supported.

To specify that functions executed with a particular connection are to be executed asynchronously, the application calls SQLSetConnectAttr and sets the SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE attribute to SQL_ASYNC_DBC_ENABLE_ON. Setting a connection attribute before establishing a connection always executes synchronously. Also, the operation setting the connection attribute SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE with SQLSetConnectAttr always executes synchronously.

An application can enable asynchronous operation before making a connection. Because the Driver Manager cannot determine which driver to use before making a connection, the Driver Manager will always return success in SQLSetConnectAttr. However, it may fail to connect if the ODBC driver does not support asynchronous operations.

In general, there can be at most one asynchronously executing function associated with a particular connection handle or statement handle. However, a connection handle can have more than one associated statement handle. If there is no asynchronous operation executing on the connection handle, an associated statement handle can execute an asynchronous operation. Similarly, you can have an asynchronous operation on a connection handle if there are no asynchronous operations in progress on any associated statement handle. An attempt to execute an asynchronous operation using a handle that is currently executing an asynchronous operation will return HY010, "Function sequence error".

If a connection operation returns SQL_STILL_EXECUTING, an application can only call the original function and the following functions for that connection handle:

- SQLCancelHandle (on the connection handle)
- SQLGetDiagField
- SQLGetDiagRec
- SQLAllocHandle (allocating ENV/DBC)
- SQLAllocHandleStd (allocating ENV/DBC)
- SQLGetEnvAttr
- SQLGetConnectAttr
- SQLDataSources
- SQLDrivers
- SQLGetInfo
- SQLGetFunctions

The application should process diagnostic records in the repeating loop of the original function. If SQLGetDiagField or SQLGetDiagRec is called when an asynchronous function is executing, it will return the current list of diagnostic records. Each time the original function call is repeated, it clears previous diagnostic records.

If a connection is being opened or closed asynchronously, the operation is complete when the application receives SQL_SUCCESS or SQL_SUCCESS_WITH_INFO in the original function call.

A new function has been added to ODBC 3.8, SQLCancelHandle. This function cancels the six connection functions (SQLBrowseConnect, SQLConnect, SQLDisconnect, SQLDriverConnect, SQLEndTran, and SQLSetConnectAttr). An application should call SQLGetFunctions to determine if the driver supports SQLCancelHandle. As with SQLCancel, if SQLCancelHandle returns success, it does not mean the operation was canceled. An application should call the original function again to determine if the operation was canceled. SQLCancelHandle lets you cancel asynchronous operations on connection
handles or statement handles. Using SQLCancelHandle to cancel an operation on a statement handle is the same as calling SQLCancel.

It is not necessary to support both SQLCancelHandle and asynchronous connection operations at the same time. A driver can support asynchronous connection operations but not SQLCancelHandle, or vice versa.

Asynchronous connection operations and SQLCancelHandle can also be used by ODBC 3.x and ODBC 2.x applications with an ODBC 3.8 driver and ODBC 3.8 Driver Manager. For information about how to enable an older application to use new features in later ODBC version, see Compatibility Matrix.

Connection Pooling

Whenever connection pooling is enabled, asynchronous operations are only minimally supported for establishing a connection (with SQLConnect and SQLDriverConnect) and closing a connection with SQLDisconnect. But an application should still be able to handle the SQL_STILL_EXECUTING return value from SQLConnect, SQLDriverConnect, and SQLDisconnect.

When connection pooling is enabled, SQLEndTran and SQLSetConnectAttr are supported for asynchronous operations.

Example

Description

The following example shows how to use SQLSetConnectAttr to enable asynchronous execution for connection-related functions.

Code

```c
BOOL AsyncConnect (SQLHANDLE hdbc)
{
  SQLRETURN r;
  SQLHANDLE hdbc;

  // Enable asynchronous execution of connection functions.
  // This must be executed synchronously, that is r != SQL_STILL_EXECUTING
  r = SQLSetConnectAttr(
    hdbc,
    SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE,
    reinterpret_cast<SQLPOINTER>(SQL_ASYNC_DBC_ENABLE_ON),
    0);
  if (r != SQL_SUCCESS && r != SQL_SUCCESS_WITH_INFO)
    return FALSE;

  TCHAR szConnStrIn[256] = _T("DSN=AsyncDemo");
  r = SQLDriverConnect(hdbc, NULL, (SQLTCHAR *) szConnStrIn, SQL_NTS, NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
  if (r == SQL_ERROR)
  {
    // Use SQLGetDiagRec to process the error.
    // If SQLState is HY114, the driver does not support asynchronous execution.
    return FALSE;
  }

  while (r == SQL_STILL_EXECUTING)
  {
    // Do something else.
    r = SQLDriverConnect(hdbc, NULL, (SQLTCHAR *) szConnStrIn, SQL_NTS, NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
  }
  if (r != SQL_SUCCESS && r != SQL_SUCCESS_WITH_INFO)
    return FALSE;

  return TRUE;
}
```

Example
Description

This example shows asynchronous commit operations. Rollback operations can also be done this way.

Code

```c
BOOL AsyncCommit ()
{
    SQLRETURN r;
    // Assume that SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE is SQL_ASYNC_DBC_ENABLE_ON.
    r = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
    while (r == SQL_STILL_EXECUTING)
    {
        // Do something else.
        r = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
    }
    if (r != SQL_SUCCESS && r != SQL_SUCCESS_WITH_INFO)
    {
        return FALSE;
    }
    return TRUE;
}
```

See Also

Executing Statements ODBC

Asynchronous Execution (Notification Method)

ODBC allows asynchronous execution of connection and statement operations. An application thread can call an ODBC function in asynchronous mode and the function can return before the operation is complete, allowing the application thread to perform other tasks. In the Windows 7 SDK, for asynchronous statement or connection operations, an application determined that the asynchronous operation was complete using the polling method. For more information, see Asynchronous Execution (Polling Method). Beginning in the Windows 8 SDK, you can determine that an asynchronous operation is complete using the notification method.

In the polling method, applications need to call the asynchronous function each time it wants the status of the operation. The notification method is similar to callback and wait in ADO.NET. ODBC, however, uses Win32 events as the notification object.

The ODBC Cursor Library and ODBC asynchronous notification cannot be used at the same time. Setting both attributes will return an error with SQLSTATE S1119 (Cursor Library and Asynchronous Notification cannot be enabled at the same time).

See Notification of Asynchronous Function Completion for information for driver developers.

Note

The notification method is not supported with cursor library. An application will receive error message if it attempts to enable cursor library via SQLSetConnectAttr, when the notification method is enabled.

Overview

When an ODBC function is called in asynchronous mode, the control is returned to the calling application immediately with the return code SQL_STILL_EXECUTING. The application must repeatedly poll the function until it returns something other than SQL_STILL_EXECUTING. The polling loop increases CPU utilization, causing poor performance in many asynchronous scenarios.

Whenever the notification model is used, the polling model is disabled. Applications should not call the original function again. Call SQLCompleteAsync Function to complete the asynchronous operation. If an application calls the original function again before the asynchronous operation is complete, the call will return SQL_ERROR with SQLSTATE IM017 (Polling is disabled in Asynchronous Notification Mode).

When using the notification model, the application can call SQLCancel or SQLCancelHandle to cancel a statement or connection operation. If the cancel request is successful, ODBC will return SQL_SUCCESS. This message does not indicate that the function was actually canceled; it indicates that the cancel request was processed. Whether the function is actually canceled is driver-dependent and data source-dependent. When an operation is canceled, the
Driver Manager will still signal the event. The Driver Manager returns SQL_ERROR in the return code buffer and the state is SQLSTATE HY008 (Operation canceled) to indicate the cancellation is successful. If the function completed its normal processing, the Driver Manager returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

Downlevel Behavior

The ODBC Driver Manager version supporting this notification on complete is ODBC 3.81.

<table>
<thead>
<tr>
<th>Application ODBC Version</th>
<th>Driver Manager Version</th>
<th>Driver Version</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>New application of any ODBC version</td>
<td>ODBC 3.81</td>
<td>ODBC 3.80 Driver</td>
<td>Application can use this feature if the driver supports this feature, otherwise the Driver Manager will error out.</td>
</tr>
<tr>
<td>New application of any ODBC version</td>
<td>ODBC 3.81</td>
<td>Pre-ODBC 3.80 Driver</td>
<td>The Driver Manager will error out if the driver does not support this feature.</td>
</tr>
<tr>
<td>New application of any ODBC version</td>
<td>Pre-ODBC 3.81</td>
<td>Any</td>
<td>When the application uses this feature, an old Driver Manager will regard the new attributes as driver-specific attributes, and the driver should error out. A new Driver Manager will not pass these attributes to the driver.</td>
</tr>
</tbody>
</table>

An application should check the Driver Manager version before using this feature. Otherwise, if a poorly written driver does not error out and the Driver Manager version is pre ODBC 3.81, behavior is undefined.

Use Cases

This section shows use cases for asynchronous execution and the polling mechanism.

Integrate Data from Multiple ODBC Sources

A data integration application asynchronously fetches data from multiple data sources. Some of the data are from remote data sources and some data are from local files. The application cannot continue until the asynchronous operations are completed.

Instead of repeatedly polling an operation to determine if it is complete, the application can create an event object and associate it with an ODBC connection handle or an ODBC statement handle. The application then calls operating system synchronization APIs to wait on one event object or many event objects (both ODBC events and other Windows events). ODBC will signal the event object when the corresponding ODBC asynchronous operation is completed.

On Windows, Win32 event objects will be used and that will provide the user a unified programming model. Driver Managers on other platforms can use the event object implementation specific to those platforms.

The following code sample demonstrates the use of connection and statement asynchronous notification:

```c
#define NUMBER_OPERATIONS 5
int AsynNotificationSample(void)
{
    RETCODE rc;
    SQLHENV hEnv = NULL;
    SQLHDBC arhDbc[NUMBER_OPERATIONS] = {NULL};
    SQLHSTMT arhStmt[NUMBER_OPERATIONS] = {NULL};
    HANDLE arhDBCEvent[NUMBER_OPERATIONS] = {NULL};
    RETCODE arrcDBC[NUMBER_OPERATIONS] = {0};
    HANDLE arhSTMTEvent[NUMBER_OPERATIONS] = {NULL};
    RETCODE arrcSTMNT[NUMBER_OPERATIONS] = {0};

    rc = SQLAllocHandle(SQL_HANDLE_ENV, NULL, &hEnv);
    if (!SQL_SUCCEEDED(rc)) goto Cleanup;

    rc = SQLSetEnvAttr(hEnv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER) SQL_OV_ODBC3_80, SQL_IS_INTEGER);
    if (!SQL_SUCCEEDED(rc)) goto Cleanup;

    // This function opens NUMBER_OPERATIONS connections and executes one query on statement of each connection.
    // Asynchronous Notification is used

    for (int i = 0; i < NUMBER_OPERATIONS; i++) {
        arhDbc[i] = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &arhDbc[i]);
        if (!SQL_SUCCEEDED(arhDbc[i])) goto Cleanup;
        rc = SQLSetEnvAttr(arhDbc[i], SQL_ATTR_ODBC_VERSION, (SQLPOINTER) SQL_OV_ODBC3_80, SQL_IS_INTEGER);
        if (!SQL_SUCCEEDED(rc)) goto Cleanup;

        arhStmt[i] = SQLAllocStmt(arhDbc[i], SQL_NTSPLATFORMOPEN, SQL_CURSOR_FORWARD_ONLY);
        if (!SQL_SUCCEEDED(arhStmt[i])) goto Cleanup;
        rc = SQLSetStmtAttr(arhStmt[i], SQL_ATTR_PREF_CURSOR, SQL_CURSOR_FORWARD_ONLY, SQL_IS_INTEGER);
        if (!SQL_SUCCEEDED(rc)) goto Cleanup;

        arhDBCEvent[i] = OpenEventThreadEvent(hEnv, SQL_DB_CREATE, SQL_GAE_CREATE, SQL_DA_CREATE);
        if (!SQL_SUCCEEDED(arhDBCEvent[i])) goto Cleanup;

        arhSTMTEvent[i] = OpenEventThreadEvent(hEnv, SQL_STMT_CREATE, SQL_GAE_CREATE, SQL_DA_CREATE);
        if (!SQL_SUCCEEDED(arhSTMTEvent[i])) goto Cleanup;

        rc = SQLSetStmtAttr(arhStmt[i], SQL_ATTR_ODBC_VERSION, (SQLPOINTER) SQL_OV_ODBC3_80, SQL_IS_INTEGER);
        if (!SQL_SUCCEEDED(rc)) goto Cleanup;

        arrcDBC[i] = SQLAllocHandle(SQL_HANDLE_STMT, arhDBCEvent[i], &arhSTMTEvent[i]);
        if (!SQL_SUCCEEDED(arrcDBC[i])) goto Cleanup;

        arrcSTMNT[i] = SQLAllocHandle(SQL_HANDLE_STMT, arhSTMTEvent[i], &arhDBCEvent[i]);
        if (!SQL_SUCCEEDED(arrcSTMNT[i])) goto Cleanup;

        rc = SQLDriverConnect(arhDBCEvent[i], arhSTMTEvent[i], sqlConnect, SQL_DRIVER_NOPROMPT);
        if (!SQL_SUCCEEDED(rc)) goto Cleanup;

        // Execute query
        rc = SQLSetStmtAttr(arhStmt[i], SQL_ATTR_EXEC模式, SQL_EXEC_MODE_EXECUTE, SQL_IS_INTEGER);
        if (!SQL_SUCCEEDED(rc)) goto Cleanup;

        // Set event
        rc = SQLSetEvent(arhDBCEvent[i], SQL_STMNT_QUERY_FINISHED, arhSTMTEvent[i], SQL_DAS_QUERY_FINISHED);
        if (!SQL_SUCCEEDED(rc)) goto Cleanup;

        // Set event
        rc = SQLSetEvent(arhDBCEvent[i], SQL_STMT_QUERY_FINISHED, arhSTMTEvent[i], SQL_DAS_STMT_QUERY_FINISHED);
        if (!SQL_SUCCEEDED(rc)) goto Cleanup;
    }

    // Wait for events
    for (int i = 0; i < NUMBER_OPERATIONS; i++) {
        arrcDBC[i] = SQLSetStmtAttr(arhDBCEvent[i], SQL_STMT_QUERY_FINISHED, SQL_DAS_STMT_QUERY_FINISHED, SQL_IS_INTEGER);
        if (!SQL_SUCCEEDED(arrcDBC[i])) goto Cleanup;
        arrcSTMNT[i] = SQLSetStmtAttr(arhSTMTEvent[i], SQL_STMT_QUERY_FINISHED, SQL_DAS_STMT_QUERY_FINISHED, SQL_IS_INTEGER);
        if (!SQL_SUCCEEDED(arrcSTMNT[i])) goto Cleanup;
    }
}

Cleanup:
return 0;
```
// Alloc NUMBER_OPERATIONS connection handles
for (int i=0; i<NUMBER_OPERATIONS; i++) {
    rc = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &arhDbc[i]);
    if (!SQL_SUCCEEDED(rc)) goto Cleanup;
}

// Enable DBC Async on all connection handles
for (int i=0; i<NUMBER_OPERATIONS; i++) {
    rc = SQLSetConnectAttr(arhDbc[i], SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE, (SQLPOINTER)SQL_ASYNC_DBC_ENABLE_ON, SQL_IS_INTEGER);
    if (!SQL_SUCCEEDED(rc)) goto Cleanup;
}

// Application must create event objects
for (int i=0; i<NUMBER_OPERATIONS; i++) {
    arhDBCEvent[i] = CreateEvent(NULL, FALSE, FALSE, NULL); // Auto-reset, initial state is not-sighnaled
    if (!arhDBCEvent[i]) goto Cleanup;
}

// Enable notification on all connection handles
// Event
for (int i=0; i<NUMBER_OPERATIONS; i++) {
    rc = SQLSetConnectAttr(arhDbc[i], SQL_ATTR_ASYNC_DBC_EVENT, arhDBCEvent[i], SQL_IS_POINTER);
    if (!SQL_SUCCEEDED(rc)) goto Cleanup;
}

// Initiate connect establishing
for (int i=0; i<NUMBER_OPERATIONS; i++) {
    SQLDriverConnect(arhDbc[i], NULL, (SQLTCHAR*)TEXT("Driver={ODBC Driver 11 for SQL Server};SERVER=dp-srv-sql2k;DATABASE=pubs;UID=sa;PWD=XYZ;"), SQL_NTS, NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
}

// Can do some other staff before calling WaitForMultipleObjects
WaitForMultipleObjects(NUMBER_OPERATIONS, arhDBCEvent, TRUE, INFINITE); // Wait All

// Complete connect API calls
for (int i=0; i<NUMBER_OPERATIONS; i++) {
    SQLCompleteAsync(SQL_HANDLE_DBC, arhDbc[i], &arrcDBC[i]);
}

BOOL fFail = FALSE; // Whether some connection opening fails.
for (int i=0; i<NUMBER_OPERATIONS; i++) {
    if (!SQL_SUCCEEDED(arrcDBC[i]))
        fFail = TRUE;
}

// If some SQLDriverConnect() fail, clean up.
if (fFail) {
    for (int i=0; i<NUMBER_OPERATIONS; i++)
        if (SQL_SUCCEEDED(arrcDBC[i]))
            SQLDisconnect(arhDbc[i]); // This is also async
        else
            SetEvent(arhDBCEvent[i]); // Previous SQLDriverConnect() failed. No need to call SQLDisconnect().
}
WaitForMultipleObjects(NUMBER_OPERATIONS, arhDBCEvent, TRUE, INFINITE);
for (int i=0; i<NUMBER_OPERATIONS; i++) {
    if (!SQL_SUCCEEDED(arrcDBC[i]))
        SQLCompleteAsync(SQL_HANDLE_DBC, arhDbc[i], &arrcDBC[i]); // To Complete
}
    goto Cleanup;
}

// Statement Operations begin here
// Alloc statement handle
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    rc = SQLAllocHandle(SQL_HANDLE_STMT, arhDbc[i], &arhStmt[i]);
    if ( !SQL_SUCCEEDED(rc) ) goto Cleanup;
}

// Enable STMT Async on all statement handles
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    rc = SQLSetStmtAttr(arhStmt[i], SQL_ATTR_ASYNC_ENABLE, (SQLPOINTER)SQL_ASYNC_ENABLE_ON, SQL_IS_INTEGER);
    if ( !SQL_SUCCEEDED(rc) ) goto Cleanup;
}

// Create event objects
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    arhSTMTEvent[i] = CreateEvent(NULL, FALSE, FALSE, NULL); // Auto-reset, initial state is not-signaled
    if (!arhSTMTEvent[i]) goto Cleanup;
}

// Enable notification on all statement handles
// Event
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    rc = SQLSetStmtAttr(arhStmt[i], SQL_ATTR_ASYNC_STMT_EVENT, arhSTMTEvent[i], SQL_IS_POINTER);
    if ( !SQL_SUCCEEDED(rc) ) goto Cleanup;
}

// Initiate SQLExecDirect() calls
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    SQLExecDirect(arhStmt[i], (SQLTCHAR*)TEXT("select t_lname, t_fname from authors"), SQL_NTS);
}

// Can do some other staff before calling WaitForMultipleObjects
WaitForMultipleObjects(NUMBER_OPERATIONS, arhSTMTEvent, TRUE, INFINITE); // Wait All

// Now, call SQLCompleteAsync to complete the operation and get return code
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    SQLCompleteAsync(SQL_HANDLE_STMT, arhStmt[i], &arrcSTMT[i]);
}

// Check return values
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    if ( !SQL_SUCCEEDED(arrcSTMT[i]) ) goto Cleanup;
}

for (int i=0; i<NUMBER_OPERATIONS; i++)
{    //Do some binding jobs here, set SQL_ATTR_ROW_ARRAY_SIZE
    SQLFetch(arhStmt[i]);
}

// Can do some other staff before calling WaitForMultipleObjects
WaitForMultipleObjects(NUMBER_OPERATIONS, arhSTMTEvent, TRUE, INFINITE);

// Now, to complete the operations and get return code
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    SQLCompleteAsync(SQL_HANDLE_STMT, arhStmt[i], &arrcSTMT[i]);
}

// Check return code
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    if ( !SQL_SUCCEEDED(arrcSTMT[i]) ) goto Cleanup;
}

// USE fetched data here!!

Cleanup:
for (int i=0; i<NUMBER_OPERATIONS; i++)
{    if (arhStmt[i])
    { SQLFreeHandle(SQL_HANDLE_STMT, arhStmt[i]);
      arhStmt[i] = NULL;
    }
Determining if a Driver Supports Asynchronous Notification

An ODBC application can determine if an ODBC driver supports asynchronous notification by calling `SQLGetInfo`. The ODBC Driver Manager will consequently call the `SQLGetInfo` of the driver with `SQL_ASYNC_NOTIFICATION`.

```c
for (int i=0; i<NUMBER_OPERATIONS; i++)
{
    if (arhSTMTEvent[i])
    {
        CloseHandle(arhSTMTEvent[i]);
        arhSTMTEvent[i] = NULL;
    }
}
for (int i=0; i<NUMBER_OPERATIONS; i++)
{
    if (arhDbc[i])
    {
        SQLFreeHandle(SQL_HANDLE_DBC, arhDbc[i]);
        arhDbc[i] = NULL;
    }
}
for (int i=0; i<NUMBER_OPERATIONS; i++)
{
    if (arhDBCEvent[i])
    {
        CloseHandle(arhDBCEvent[i]);
        arhDBCEvent[i] = NULL;
    }
}
if (hEnv)
{
    SQLFreeHandle(SQL_HANDLE_ENV, hEnv);
    hEnv = NULL;
}
return 0;
```

```c
SQLUINTEGER InfoValue;
SQLLEN cInfLength;
SQLRETURN retcode;
retcode = SQLGetInfo (hDbc, SQL_ASYNC_NOTIFICATION, &InfoValue, sizeof(InfoValue), NULL);
if (SQL_SUCCEEDED(retcode))
{
    if (SQL_ASYNC_NOTIFICATION_CAPABLE == InfoValue)
    {
        // The driver supports asynchronous notification
    }
    else if (SQL_ASYNC_NOTIFICATION_NOT_CAPABLE == InfoValue)
    {
        // The driver does not support asynchronous notification
    }
}
```

Associating a Win32 Event Handle with an ODBC Handle

Applications are responsible for creating Win32 event objects using the corresponding Win32 functions. An application can associate one Win32 event handle with one ODBC connection handle or one ODBC statement handle.

Connection attributes `SQL_ATTR_ASYNC_DBC_FUNCTION_ENABLE` and `SQL_ATTR_ASYNC_DBC_EVENT` determine whether ODBC executes in asynchronous mode and whether ODBC enables notification mode for a connection handle. Statement attributes `SQL_ATTR_ASYNC_ENABLE` and `SQL_ATTR_ASYNC_STMT_EVENT` determine whether ODBC executes in asynchronous mode and whether ODBC enables notification mode for a statement handle.
An application can temporally disable asynchronous operation mode. ODBC ignores values of SQL_ATTR_ASYNC_DBC_EVENT if the connection level asynchronous operation is disabled. ODBC ignores values of SQL_ATTR_ASYNC_STMT_EVENT if the statement level asynchronous operation is disabled.

### Synchronous call of SQLSetStmtAttr and SQLSetConnectAttr
- **SQLSetConnectAttr** supports asynchronous operations but the invocation of SQLSetConnectAttr to set SQL_ATTR_ASYNC_DBC_EVENT is always synchronous.
- **SQLSetStmtAttr** does not support asynchronous execution.

### Error-out scenario
When SQLSetConnectAttr is called before making a connection, the Driver Manager cannot determine which driver to use. Therefore, the Driver Manager returns success for SQLSetConnectAttr but the attribute may not be ready to set in the driver. The Driver Manager will set these attributes when the application calls a connection function. The Driver Manager may error-out because driver does not support asynchronous operations.

### Inheritance of connection attributes
Usually, the statements of a connection will inherit the connection attributes. However, the attribute SQL_ATTR_ASYNC_DBC_EVENT is not inheritable and only affects the connection operations.

To associate an event handle with an ODBC connection handle, an ODBC application calls ODBC API SQLSetConnectAttr and specifies SQL_ATTR_ASYNC_DBC_EVENT as the attribute and the event handle as the attribute value. The new ODBC attribute SQL_ATTR_ASYNC_DBC_EVENT is of type SQL_IS_POINTER.

```c
HANDLE hEvent;
hEvent = CreateEvent( 
    NULL, // default security attributes
    FALSE, // auto-reset event
    FALSE, // initial state is non-signaled
    NULL  // no name
);
```

Usually, applications create auto-reset event objects. ODBC will not reset the event object. Applications must make sure that the object is not in signaled state before calling any asynchronous ODBC function.

```c
SQLRETURN retcode;
retcode = SQLSetConnectAttr( 
    hDBC, 
    SQL_ATTR_ASYNC_DBC_EVENT, // Attribute name
    (SQLPOINTER) hEvent,      // Win32 Event handle
    SQL_IS_POINTER);          // Length Indicator
```

SQL_ATTR_ASYNC_DBC_EVENT is a Driver Manager-only attribute that will not be set in the driver.

The default value of SQL_ATTR_ASYNC_DBC_EVENT is NULL. If the driver does not support asynchronous notification, getting or setting SQL_ATTR_ASYNC_DBC_EVENT will return SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier).

If the last SQL_ATTR_ASYNC_DBC_EVENT value set on an ODBC connection handle is not NULL and the application enabled asynchronous mode by setting attribute SQL_ATTR_ASYNC_DBC_FUNCTION_ENABLE with SQL_ATTR_ASYNC_DBC_ENABLE_ON, calling any ODBC connection function that supports asynchronous mode will get a completion notification. If the last SQL_ATTR_ASYNC_DBC_EVENT value set on an ODBC connection handle is NULL, ODBC will not send the application any notification, regardless whether asynchronous mode is enabled.

An application can set SQL_ATTR_ASYNC_DBC_EVENT before or after setting the attribute SQL_ATTR_ASYNC_DBC_FUNCTION_ENABLE.

Applications can set the SQL_ATTR_ASYNC_DBC_EVENT attribute on an ODBC connection handle before calling a connection function (SQLConnect, SQLBrowseConnect, or SQLDriverConnect). Because the ODBC Driver Manager does not know which ODBC driver the application will use, it will return SQL_SUCCESS. When the application calls a connection function, the ODBC Driver Manager will check whether the driver supports asynchronous notification. If the driver does not support asynchronous notification, the ODBC Driver Manager will return SQL_ERROR with SQLSTATE 51_118 (Driver does not support asynchronous notification). If the driver supports asynchronous notification, the ODBC Driver Manager will call the driver and set the corresponding attributes SQL_ATTR_ASYNC_DBC_NOTIFICATION_CALLBACK and SQL_ATTR_ASYNC_DBC_NOTIFICATION_CONTEXT.

Similarly, an application calls SQLSetStmtAttr on an ODBC statement handle and specifies the SQL_ATTR_ASYNC_STMT_EVENT attribute to enable or disable statement level asynchronous notification. Because a statement function is always called after the connection is established, SQLSetStmtAttr will return SQL_ERROR with SQLSTATE S1_118 (Driver does not support asynchronous notification) immediately if the corresponding driver does not support asynchronous operations or the driver supports asynchronous operation but does not support asynchronous notification.
SQL_ATTR_ASYNC_STMT_EVENT, which can be set to NULL, is a Driver Manager-only attribute that will not be set in the driver.

The default value of SQL_ATTR_ASYNC_STMT_EVENT is NULL. If the driver does not support asynchronous notification, getting or setting the SQL_ATTR_ASYNC_STMT_EVENT attribute will return SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier).

An application should not associate the same event handle with more than one ODBC handle. Otherwise, one notification will be lost if two asynchronous ODBC function invocations complete on two handles that share the same event handle. To avoid a statement handle inheriting the same event handle from the connection handle, ODBC returns SQL_ERROR with SQLSTATE IM016 (Cannot set statement attribute into connection handle) if an application sets SQL_ATTR_ASYNC_STMT_EVENT on a connection handle.

Calling Asynchronous ODBC Functions

After enabling asynchronous notification and starting an asynchronous operation, the application can call any ODBC function. If the function belongs to the set of functions that support asynchronous operation, the application will get a completion notification when the operation completes, regardless of whether the function failed or succeeded. The only exception is that the application calls an ODBC function with an invalid connection or statement handle. In this case, ODBC will not get the event handle and set it to the signaled state.

The application must ensure that the associated event object is in a non-signaled state before starting an asynchronous operation on the corresponding ODBC handle. ODBC will not reset the event object.

Getting Notification from ODBC

An application thread can call WaitForSingleObject to wait on one event handle or call WaitForMultipleObjects to wait on an array of event handles and be suspended until one or all of the event objects become signaled or the time-out interval elapses.

Freeing a Statement Handle ODBC

As mentioned earlier, it is more efficient to reuse statements than to drop them and allocate new ones. Before executing a new SQL statement on a statement, applications should be sure 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 need to 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. After freeing the statement, it is an application programming error to use the statement's handle in a call to an ODBC function; doing so has undefined but probably fatal consequences.

When SQLFreeHandle is called, the driver releases the structure used to store information about the statement.

Retrieving Results (Basic)
A result set is a set of rows on the data source that matches certain criteria. It is a conceptual table that results from a query and that is available to an application in tabular form. SELECT statements, catalog functions, and some procedures create result sets. In the following example, the first SQL statement creates a result set containing all the rows and all the columns in the Orders table, and the second SQL statement creates a result set containing OrderID, SalesPerson, and Status columns for the rows in the Orders table in which the Status is OPEN:

```
SELECT * FROM Orders
SELECT OrderID, SalesPerson, Status FROM Orders WHERE Status = 'OPEN'
```

A result set can be empty, which is different from no result set at all. For example, the following SQL statement creates an empty result set:

```
SELECT * FROM Orders WHERE 1 = 2
```

An empty result set is no different from any other result set except that it has no rows. For example, the application can retrieve metadata for the result set, can attempt to fetch rows, and must close the cursor over the result set.

The process of retrieving rows from the data source and returning them to the application is called fetching. This section explains the basic parts of that process. For information about more advanced topics, such as block and scrollable cursors, see Block Cursors and Scrollable Cursors. For information about updating, deleting, and inserting rows, see Updating Data Overview.

This section contains the following topics.

- Was a Result Set Created?
- Result Set Metadata
- Binding Columns
- Fetching Data
- Closing the Cursor

### Was a Result Set Created?

In most situations, application programmers know whether the statements their application executes will create a result set. This is the case if the application uses hard-coded SQL statements written by the programmer. It is usually the case when the application constructs SQL statements at run time: The programmer can easily include code that flags whether a SELECT statement or an INSERT statement is being constructed. In a few situations, the programmer cannot possibly know whether a statement will create a result set. This is true if the application provides a way for the user to enter and execute an SQL statement. It is also true when the application constructs a statement at run time to execute a procedure.

In such cases, the application calls SQLNumResultCols to determine the number of columns in the result set. If this is 0, the statement did not create a result set; if it is any other number, the statement did create a result set.

The application can call SQLNumResultCols at any time after the statement is prepared or executed. However, because some data sources cannot easily describe the result sets that will be created by prepared statements, performance will suffer if SQLNumResultCols is called after a statement is prepared but before it is executed.

Some data sources also support determining the number of rows that an SQL statement returns in a result set. To do so, the application calls SQLRowCount. Exactly what the row count represents is indicated by the setting of the SQL_DYNAMIC_CURSOR_ATTRIBUTES2, SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2, SQL_KEYSET_CURSOR_ATTRIBUTES2, or SQL_STATIC_CURSOR_ATTRIBUTES2 option (depending on the type of the cursor) returned by a call to SQLGetInfo. This bitmask indicates for each cursor type whether the row count returned is exact, approximate, or is not available at all. Whether row counts for static or keyset-driven cursors are affected by changes made through SQLBulkOperations or SQLSetPos, or by positioned update or delete statements, depends on other bits returned by the same option arguments listed previously. For more information, see the SQLGetInfo function description.

### Result Set Metadata

Metadata is data that describes other data. For example, result set metadata describes the 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.

Interoperable applications should always check the metadata of result set columns. The metadata for a column in a result set might differ from the metadata for the column as returned by a catalog function. For example, suppose that an updatable column is included in a result set created by joining two tables. While SQLColumnPrivileges might indicate that a user can update the column, the result set metadata might not if the column is on the "many" side of the join; many data sources can update columns on the "one" side of a join but not on the "many" side. Even data types cannot be assumed to be the same, because the data source might promote the data type while creating the result set.

This section contains the following topics.
How is Metadata Used?

Applications require metadata 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 metadata for a column depends on the type of the application.

Vertical applications work with predefined tables and perform predefined operations on those tables. Because the result set metadata for such applications is defined before the application is 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 metadata is hard-coded in the application, a change to the tables used by the application generally implies a change to the application code. This is rarely a problem, because such changes are usually made as part of a new release of the application.

Like vertical applications, custom applications generally work with predefined tables and perform predefined operations on those tables. For example, an application might be written to transfer data among three different data sources; the data to be transferred is usually known when the application is written. Thus, custom applications also tend to have hard-coded metadata.

Generic applications, especially those that support ad hoc queries, almost never know the metadata of the result sets they create. Therefore, they must discover the metadata at run time using the functions `SQLNumResultCols`, `SQLDescribeCol`, and `SQLColAttribute`, which are described in the next section, `SQLDescribeCol` and `SQLColAttribute`.

All applications, regardless of their type, can hard-code metadata for the result sets returned by the catalog functions. These result sets are defined in the reference section of this manual.

SQLDescribeCol and SQLColAttribute

`SQLDescribeCol` and `SQLColAttribute` are used to retrieve result set metadata. The difference between these two functions is that `SQLDescribeCol` always returns the same five pieces of information (a column's name, data type, precision, scale, and nullability), while `SQLColAttribute` returns a single piece of information requested by the application. However, `SQLColAttribute` can return a much richer selection of metadata, including a column's case-sensitivity, display size, updatability, and searchability.

Many applications, especially ones that only display data, require only the metadata returned by `SQLDescribeCol`. For these applications, it is faster to use `SQLDescribeCol` than `SQLColAttribute` because the information is returned in a single call. Other applications, especially ones that update data, require the additional metadata returned by `SQLColAttribute` and therefore use both functions. In addition, `SQLColAttribute` supports driver-specific metadata; for more information, see `Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes`.

An application can retrieve result set metadata at any time after a statement has been prepared or executed and before the cursor over the result set is closed. Very few applications require result set metadata after the statement is prepared and before it is executed. If possible, applications should wait to retrieve metadata until after the statement is executed, because some data sources cannot return metadata for prepared statements and emulating this capability in the driver is often a slow process. For example, the driver might generate a zero-row result set by replacing the `WHERE` clause of a `SELECT` statement with the clause `WHERE 1 = 2` and executing the resulting statement.

Metadata is often expensive to retrieve from the data source. Because of this, drivers should cache any metadata they retrieve from the server and hold it for as long as the cursor over the result set is open. Also, applications should request only the metadata they absolutely need.

Binding Columns

Data fetched from the data source is returned to the application in variables that the application has allocated for this purpose. Before this can be done, the application must associate, or bind, these variables to the columns of the result set; conceptually, this process is the same as binding application variables to statement parameters. When the application binds a variable to a result set column, it describes that variable — address, data type, and so on — to the driver. The driver stores this information in the structure it maintains for that statement and uses the information to return the value from the column when the row is fetched.

This section contains the following topics.

- Binding Result Set Columns
- Using SQLBindCol
Binding Result Set Columns

Applications can bind as many or as few columns of the result set as they choose, including binding no columns at all. When a row of data is fetched, the driver returns the data for the bound columns to the application. Whether the application binds all of the columns in the result set depends on the application. For example, applications that generate reports usually have a fixed format; such applications create a result set containing all of the columns used in the report and then bind and retrieve the data for all of these columns. Applications that display screens full of data sometimes allow the user to decide which columns to display; such applications create a result set containing all columns the user might want, but bind and retrieve the data only for those columns chosen by the user.

Data can be retrieved from unbound columns by calling SQLGetData. This is commonly called to retrieve long data, which often exceeds the length of a single buffer and must be retrieved in parts.

Columns can be bound at any time, even after rows have been fetched. However, the new bindings do not take effect until the next time a row is fetched; they are not applied to data from rows already fetched.

A variable remains bound to a column until a different variable is bound to the column, until the column is unbound by calling SQLBindCol with a null pointer as the variable's address, until all columns are unbound by calling SQLFreeStmt with the SQL_UNBIND option, or until the statement is released. For this reason, the application must be sure that all bound variables remain valid as long as they are bound. For more information, see Allocating and Freeing Buffers.

Because column bindings are just information associated with the statement structure, they can be set in any order. They are also independent of the result set. For example, suppose an application binds the columns of the result set generated by the following SQL statement:

```
SELECT * FROM Orders
```

If the application then executes the SQL statement

```
SELECT * FROM Lines
```

on the same statement handle, the column bindings for the first result set are still in effect because those are the bindings stored in the statement structure. In most cases, this is a poor programming practice and should be avoided. Instead, the application should call SQLFreeStmt with the SQL_UNBIND option to unbind all the old columns and then bind new ones.

© 2016 Microsoft

Using SQLBindCol

The application binds columns by calling SQLBindCol. This function binds one column at a time. With it, the application specifies the following:

- The column number. Column 0 is the bookmark column; this column is not included in some result sets. All other columns are numbered starting with the number 1. It is an error to bind a higher-numbered column than there are columns in the result set; this error cannot be detected until the result set has been created, so it is returned by SQLFetch, not SQLBindCol.

- The C data type, address, and byte length of the variable bound to the column. It is an error to specify a C data type to which the SQL data type of the column cannot be converted; this error might not be detected until the result set has been created, so it is returned by SQLFetch, not SQLBindCol. For a list of supported conversions, see Converting Data from SQL to C Data Types in Appendix D: Data Types. For information about the byte length, see Data Buffer Length.

- The address of a length/indicator buffer. The length/indicator buffer is optional. It is used to return the byte length of binary or character data or return SQL_NULL_DATA if the data is NULL. For more information, see Using Length/Indicator Values.

When SQLBindCol is called, the driver associates this information with the statement. When each row of data is fetched, it uses the information to place the data for each column in the bound application variables.

For example, the following code binds variables to the SalesPerson and CustID columns. Data for the columns will be returned in SalesPerson and CustID.

```
Because SalesPerson is a character buffer, the application specifies its byte length (11) so that the driver can determine whether to truncate the data. The byte length of the returned title, or whether it is NULL, will be returned in SalesPersonLenOrInd.

Because CustID is an integer variable and has fixed length, there is no need to specify its byte length; the driver assumes it is sizeof(SQLINTEGER). The byte length of the returned customer ID data, or whether it is NULL, will be returned in CustIDInd. Note that the application is interested only in whether the salary is NULL, because the byte length is always sizeof(SQLINTEGER).

```
The following code executes a SELECT statement entered by the user and prints each row of data in the result set. Because the application cannot predict the shape of the result set created by the SELECT statement, it cannot bind hard-coded variables to the result set as in the preceding example. Instead, the application allocates a buffer that holds the data and a length/indicator buffer for each column in that row. For each column, it calculates the offset to the start of the memory for the column and adjusts this offset so that the data and length/indicator buffers for the column start on alignment boundaries. It then binds the memory starting at the offset to the column. From the driver’s point of view, the address of this memory is indistinguishable from the address of a variable bound in the preceding example. For more information about alignment, see Alignment.

```c
// CustID column.
SQLBindCol(hstmt, 1, SQL_C_CHAR, SalesPerson, sizeof(SalesPerson), &SalesPersonLenOrInd);
SQLBindCol(hstmt, 2, SQL_C_ULONG, &CustID, 0, &CustIDInd);
// Execute a statement to get the sales person/customer of all orders.
SQLExecDirect(hstmt, "SELECT SalesPerson, CustID FROM Orders ORDER BY SalesPerson", SQL_NTS);
// Fetch and print the data. Print "NULL" if the data is NULL. Code to
// check if rc equals SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
  if (SalesPersonLenOrInd == SQL_NULL_DATA)
    printf("NULL ");
  else
    printf("%s", SalesPerson);
  if (CustIDInd == SQL_NULL_DATA)
    printf("NULL\n");
  else
    printf("%d\n", CustID);
}
// Close the cursor.
SQLCloseCursor(hstmt);
```

The following code executes a SELECT statement entered by the user and prints each row of data in the result set. Because the application cannot predict the shape of the result set created by the SELECT statement, it cannot bind hard-coded variables to the result set as in the preceding example. Instead, the application allocates a buffer that holds the data and a length/indicator buffer for each column in that row. For each column, it calculates the offset to the start of the memory for the column and adjusts this offset so that the data and length/indicator buffers for the column start on alignment boundaries. It then binds the memory starting at the offset to the column. From the driver’s point of view, the address of this memory is indistinguishable from the address of a variable bound in the preceding example. For more information about alignment, see Alignment.
Fetching Data

The process of retrieving rows from the result set and returning them to the application is called fetching. This section describes how to fetch data.

This section contains the following topics.

- Cursors
- Fetching a Row of Data
- Getting Long Data

© 2016 Microsoft

Cursors

An application fetches data with a cursor. A cursor is different from a result set: A result set is the set of rows that matches particular search criteria, whereas a cursor is the software that returns those rows to the application. The name cursor, as it applies to databases, probably originated from the blinking cursor on a computer terminal. Just as that cursor indicates the current position on the screen and where the typed words will appear next, a cursor on a result set indicates the current position in the result set and what row will be returned next.

The cursor model in ODBC is based on the cursor model in embedded SQL. One notable difference between these models is the way cursors are opened. In embedded SQL, a cursor must be explicitly declared and opened before it can be used. In ODBC, a cursor is implicitly opened when a statement that creates a result set is executed. When the cursor is opened, it is positioned before the first row of the result set. In both embedded SQL and ODBC, a cursor must be closed after the application has finished using it.

Different cursors have different characteristics. The most common type of cursor, which is called a forward-only cursor, can only move forward through the result set. To return to a previous row, the application must close and reopen the cursor and then read rows from the beginning of the result set until it
reaches the required row. Forward-only cursors provide a fast mechanism for making a single pass through a result set.

Forward-only cursors are less useful for screen-based applications, in which the user scrolls backward and forward through the data. Such applications can use a forward-only cursor by reading the result set once, caching the data locally, and performing scrolling themselves. However, this works well only with small amounts of data. A better solution is to use a scrollable cursor, which provides random access to the result set. Such applications can also increase performance by fetching more than one row of data at a time, using what is called a block cursor. For more information about block cursors, see Using Block Cursors.

The forward-only cursor is the default cursor type in ODBC and is discussed in the following sections. For more information about block cursors and scrollable cursors, see Block Cursors and Scrollable Cursors.

### Important

Committing or rolling back a transaction, either by explicitly calling `SQLEndTran` or by operating in auto-commit mode, causes some data sources to close all the cursors on all statements on a connection. For more information, see the `SQL_CURSOR_COMMIT_BEHAVIOR` and `SQL_CURSOR_ROLLBACK_BEHAVIOR` attributes in the `SQLGetInfo` function description.

© 2016 Microsoft

## Fetching a Row of Data

To fetch a row of data, an application calls `SQLFetch`. `SQLFetch` can be called with any kind of cursor, but it only moves the rowset cursor in a forward-only direction. `SQLFetch` advances the cursor to the next row and returns the data for any columns that were bound with calls to `SQLBindCol`. When the cursor reaches the end of the result set, `SQLFetch` returns `SQL_NO_DATA`. For examples of calling `SQLFetch`, see Using Block Cursors.

Exactly how `SQLFetch` is implemented is driver-specific, but the general pattern is for the driver to retrieve the data for any bound columns from the data source, convert it according to the types of the bound variables, and place the converted data in those variables. If the driver cannot convert any data, `SQLFetch` returns an error. The application can continue fetching rows, but the data for the current row is lost. What happens to the data for unbound columns depends on the driver, but most drivers either retrieve and discard it or never retrieve it at all.

The driver also sets the values of any length/indicator buffers that have been bound. If the data value for a column is NULL, the driver sets the corresponding length/indicator buffer to `SQL_NULL_DATA`. If the data value is not NULL, the driver sets the length/indicator buffer to the byte length of the data after conversion. If this length cannot be determined, as is sometimes the case with long data that is retrieved by more than one function call, the driver sets the length/indicator buffer to `SQL_NO_TOTAL`. For fixed-length data types, such as integers and date structures, the byte length is the size of the data type.

For variable-length data, such as character and binary data, the driver checks the byte length of the converted data against the byte length of the buffer bound to the column; the buffer’s length is specified in the `BufferLength` argument in `SQLBindCol`. If the byte length of the converted data is greater than the byte length of the buffer, the driver truncates the data to fit in the buffer, returns the untruncated length in the length/indicator buffer, returns `SQL_SUCCESS_WITH_INFO`, and places SQLSTATE 01004 (Data truncated) in the diagnostics. The only exception to this is if a variable-length bookmark is truncated when returned by `SQLFetch`, which returns SQLSTATE 22001 (String data, data truncated).

Fixed-length data is never truncated, because the driver assumes that the size of the bound buffer is the size of the data type. Data truncation tends to be rare, because the application usually binds a buffer large enough to hold the entire data value; it determines the necessary size from the metadata. However, the application might explicitly bind a buffer it knows to be too small. For example, it might retrieve and display the first 20 characters of a part description or the first 100 characters of a long text column.

Character data must be null-terminated by the driver before it is returned to the application, even if it has been truncated. The null-termination character is not included in the returned byte length but does require space in the bound buffer. For example, suppose an application uses strings composed of character data in the ASCII character set, a driver has 50 characters of data to return, and the application’s buffer is 25 bytes long. In the application’s buffer, the driver returns the first 24 characters followed by a null-termination character. In the length/indicator buffer, it returns a byte length of 50.

The application can restrict the number of rows in the result set by setting the `SQL_ATTR_MAX_ROWS` statement attribute before executing the statement that creates the result set. For example, the preview mode in an application used to format reports needs only enough data to display the first page of the report. By restricting the size of the result set, such a feature would run faster. This statement attribute is intended to reduce network traffic and might not be supported by all drivers.

© 2016 Microsoft

## Getting Long Data

DBMSs define long data as any character or binary data over a certain size, such as 255 characters. This data may be small enough to be stored in a single buffer, such as a part description of several thousand characters. However, it might be too long to store in memory, such as long text documents or bitmaps. Because such data cannot be stored in a single buffer, it is retrieved from the driver in parts with `SQLGetData` after the other data in the row has been fetched.

### Note

An application can actually retrieve any type of data with `SQLGetData`, not just long data, although only character and binary data can be retrieved in parts. However, if the data is small enough to fit in a single buffer, there is generally no reason to use `SQLGetData`. It is much easier to bind a buffer to the column and let the driver return the data in the buffer.

To retrieve long data from a column, an application first calls `SQLFetchScroll` or `SQLFetch` to move to a row and fetch the data for bound columns. The application then calls `SQLGetData`. `SQLGetData` has the same arguments as `SQLBindCol`: a statement handle; a column number; the C data type, address, and byte length of an application variable; and the address of a length/indicator buffer. Both functions have the same arguments because they perform essentially the same task: They both describe an application variable to the driver and specify that the data for a particular column should be returned in that variable. The major differences are that `SQLGetData` is called after a row is fetched (and is sometimes referred to as late binding for this reason) and that the
Regarding a single column, SQLGetData behaves like SQLFetch: It retrieves the data for the column, converts it to the type of the application variable, and returns it in that variable. It also returns the byte length of the data in the length/indicator buffer. For more information about how SQLFetch returns data, see Fetching a Row of Data.

SQLGetData differs from SQLFetch in one important respect. If it is called more than once in succession for the same column, each call returns a successive part of the data. Each call except the last call returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01004 (String data, right truncated); the last call returns SQL_NO_DATA. The application is responsible for putting the long data together, which might mean concatenating the parts of the data. Each part is null-terminated; the application must remove the null-termination character if concatenating the parts. Retrieving data in parts can be done for variable-length bookmarks as well as for other long data. The value returned in the length/indicator buffer decreases in each call by the number of bytes returned in the previous call, although it is common for the driver to be unable to discover the amount of available data and return a byte length of SQL_NO_TOTAL. For example:

```c
// Declare a binary buffer to retrieve 5000 bytes of data at a time.
SQLCHAR  BinaryPtr[5000];
SQLUINTEGER  PartID;
SQLINTEGER  PartIDInd, BinaryLenOrInd, NumBytes;
SQLRETURN  rc;
SQLHSTMT  hstmt;

// Create a result set containing the ID and picture of each part.
SQLExecDirect(hstmt, "SELECT PartID, Picture FROM Pictures", SQL_NTS);

// Bind PartID to the PartID column.
SQLBindCol(hstmt, 1, SQL_C_ULONG, &PartID, 0, &PartIDInd);

// Retrieve and display each row of data.
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
  // Display the part ID and initialize the picture.
  DisplayID(PartID, PartIDInd);
  InitPicture();

  // Retrieve the picture data in parts. Send each part and the number
  // of bytes in each part to a function that displays it. The number
  // of bytes is always 5000 if there were more than 5000 bytes
  // available to return (cbBinaryBuffer > 5000). Code to check if
  // rc equals SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
  while ((rc = SQLGetData(hstmt, 2, SQL_C_BINARY, BinaryPtr, sizeof(BinaryPtr),
         &BinaryLenOrInd)) != SQL_NO_DATA) {
    NumBytes = (BinaryLenOrInd > 5000) || (BinaryLenOrInd == SQL_NO_TOTAL) ?
              5000 : BinaryLenOrInd;
    DisplayNextPictPart(BinaryPtr, NumBytes);
  }
}
// Close the cursor.
SQLCloseCursor(hstmt);
```

There are several restrictions on using SQLGetData. Generally, columns accessed with SQLGetData:

- Must be accessed in order of increasing column number (because of the way the columns of a result set are read from the data source). For example, it is an error to call SQLGetData for column 5 and then call it for column 4.
- Cannot be bound. Must have a higher column number than the last bound column. For example, if the last bound column is column 3, it is an error to call SQLGetData for column 2. For this reason, applications should make sure to place long data columns at the end of the select list.
- Cannot be used if SQLFetch or SQLFetchScroll was called to retrieve more than one row. For more information, see Using Block Cursors.

Some drivers do not enforce these restrictions. Interoperable applications should either assume they exist or determine which restrictions are not enforced by calling SQLGetInfo with the SQL_GETDATA_EXTENSIONS option.

If the application does not need all the data in a character or binary data column, it can reduce network traffic in DBMS-based drivers by setting the SQL_ATTR_MAX_LENGTH statement attribute before executing the statement. This restricts the number of bytes of data that will be returned for any character or binary column, for example. Suppose a column contains long text documents. An application that browses the table containing this column might have to display only the first page of each document. Although this statement attribute can be simulated in the driver, there is no reason to do this. In particular, if an application wants to truncate character or binary data, it should bind a small buffer to the column with SQLBindCol and let the driver truncate the data.

Closing the Cursor

When an application has finished using a cursor, it calls SQLCloseCursor to close the cursor. For example:

```c
```
Until the application closes the cursor, the statement on which the cursor is opened cannot be used for most other operations, such as executing another SQL statement. For a complete list of functions that can be called while a cursor is open, see Appendix B: ODBC State Transition Tables.

Note

To close a cursor, an application should call SQLCloseCursor, not SQLCancel. Cursors remain open until they are explicitly closed, except when a transaction is committed or rolled back, in which case some data sources close the cursor. In particular, reaching the end of the result set, when SQLFetch returns SQL_NO_DATA, does not close a cursor. Even cursors on empty result sets (result sets created when a statement executed successfully but which returned no rows) must be explicitly closed.

© 2016 Microsoft

Retrieving Results (Advanced)

An application can specify that an offset is added to bound data buffer addresses and the corresponding length/indicator buffer addresses when SQLBulkOperations, SQLFetch, SQLFetchScroll, or SQLSetPos is called. The results of these additions determine the addresses used in these operations.

Bind offsets allow an application to change bindings without calling SQLBindCol for previously bound columns. A call to SQLBindCol to rebind data changes the buffer address and the length/indicator pointer. Rebinding with an offset, on the other hand, simply adds an offset to the existing bound data buffer address and length/indicator buffer address. When offsets are used, the bindings are a "template" of how the application buffers are laid out and the application can move this "template" to different areas of memory by changing the offset. A new offset can be specified at any time and is always added to the originally bound values.

To specify a bind offset, the application sets the SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute to the address of an SQLINTEGER buffer. Before the application calls a function that uses the bindings, such as SQLBulkOperations, SQLFetch, SQLFetchScroll, or SQLSetPos, it places an offset in bytes in this buffer, as long as neither the data buffer address nor the length/indicator buffer address is 0, and as long as the bound column is in the result set. The sum of the address and the offset must be a valid address. (This means that either or both the offset and the address to which the offset is added can be invalid, as long as their sum is a valid address.) The SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute is a pointer so that the offset value can be applied to more than one set of binding data, all of which can be changed by changing one offset value. An application must make sure that the pointer remains valid until the cursor is closed.

Note

Binding offsets are not supported by ODBC 2.x drivers.

This section contains the following topics.

- Block Cursors
- Scannable Cursors
- The ODBC Cursor Library
- Multiple Results

© 2016 Microsoft

Block Cursors

Many applications spend a significant amount of time bringing data across the network. Part of this time is spent actually bringing the data across the network, and part of it is spent on network overhead, such as the call made by the driver to request a row of data. The latter time can be reduced if the application makes efficient use of block, or fat, cursors, which can return more than one row at a time.

An application always has the option of using a block cursor. On data sources from which only one row at a time can be fetched, block cursors must be simulated in the driver. This can be done by performing multiple single-row fetches. While this is unlikely to provide any performance gains, it opens opportunities for applications. Such applications will then experience performance increases as DBMSs implement block cursors natively and the drivers associated with those DBMSs expose them.

The rows returned in a single fetch with a block cursor are called the rowset. It is important not to confuse the rowset with the result set. The result set is maintained at the data source, while the rowset is maintained in application buffers. While the result set is fixed, the rowset is not — it changes position and contents each time a new set of rows is fetched. Just as a single-row cursor such as the traditional SQL forward-only cursor points to a current row, a block cursor points to the rowset, which can be thought of as current rows.

To perform operations that operate on a single row when multiple rows have been fetched, the application must first indicate which row is the current row. The current row is required by calls to SQLGetData and positioned update and delete statements. When a block cursor first returns a rowset, the current row is the first row of the rowset. To change the current row, the application calls SQLSetPos or SQLBulkOperations (to update by bookmark). The following illustration shows the relationship of the result set, rowset, current row, rowset cursor, and block cursor. For more information, see Using Block Cursors, later in this section, and Positioned Update and Delete Statements and Updating Data with SQLSetPos.
Whether a cursor is a block cursor is independent of whether it is scrollable. For example, most of the work in a report application is spent retrieving and printing rows. Because of this, it will work fastest with a forward-only, block cursor. It uses a forward-only cursor to avoid the expense of a scrollable cursor, and a block cursor to reduce the network traffic.

This section contains the following topics:

- Binding Columns for Use with Block Cursors
- Using Block Cursors
- Row Status Array

© 2016 Microsoft

**Binding Columns for Use with Block Cursors**

Because block cursors return multiple rows, applications that use them must bind an array of variables to each column instead of a single variable. These arrays are collectively known as the rowset buffers. Following are the two styles of binding:

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

As when the application binds single variables to columns, it calls SQLBindCol to bind arrays to columns. The only difference is that the addresses passed are array addresses, not single variable addresses. The application sets the SQL_BIND_BY_COLUMN statement attribute to specify whether it is using column-wise or row-wise binding is largely a matter of application preference. Row-wise binding might correspond more closely to the application’s layout of data, in which case it would provide better performance.

This section contains the following topics:

- Column-Wise Binding
- Row-Wise Binding

© 2016 Microsoft

**Column–Wise Binding**

When using column-wise binding, an application binds one or two, or in some cases three, arrays to each column for which data is to be returned. The first array holds the data values, and the second array holds length/indicator buffers. Indicators and length values can be stored in separate buffers by setting the SQL_DESC_INDICATOR_PTR and SQL_DESC_OCTET_LENGTH_PTR descriptor fields to different values; if this is done, a third array is bound. Each array contains as many elements as there are rows in the rowset.

The application declares that it is using column-wise binding with the SQL_ATTR_ROW_BIND_TYPE statement attribute, which determines the bind type for rowset buffers as opposed to parameter set buffers. The driver returns the data for each row in successive elements of each array. The following illustration shows how column-wise binding works.
For example, the following code binds 10-element arrays to the OrderID, SalesPerson, and Status columns:

```c
#define ROW_ARRAY_SIZE 10

SQLINTEGER OrderIDArray[ROW_ARRAY_SIZE], NumRowsFetched;
SQLCHAR SalesPersonArray[ROW_ARRAY_SIZE][11], StatusArray[ROW_ARRAY_SIZE][7];

SQLINTEGER OrderIDIndArray[ROW_ARRAY_SIZE],
SalesPersonLenOrIndArray[ROW_ARRAY_SIZE], StatusLenOrIndArray[ROW_ARRAY_SIZE];

SQLUSMALLINT RowStatusArray[ROW_ARRAY_SIZE], i;
SQLRETURN rc;
SQLHSTMT hstmt;

// Set the SQL_ATTR_ROW_BIND_TYPE statement attribute to use
// column-wise binding. Declare the rowset size with the
// SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Set the
// SQL_ATTR_ROW_STATUS_PTR statement attribute to point to the
// row status array. Set the SQL_ATTR_ROWS_FETCHED_PTR statement
// attribute to point to cRowsFetched.
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, ROW_ARRAY_SIZE, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROWS_FETCHED_PTR, &NumRowsFetched, 0);

// Bind arrays to the OrderID, SalesPerson, and Status columns.
SQLBindCol(hstmt, 1, SQL_C_ULONG, OrderIDArray, 0, OrderIDIndArray);
SQLBindCol(hstmt, 2, SQL_C_CHAR, SalesPersonArray, sizeof(SalesPersonArray[0]),
SalesPersonLenOrIndArray);
SQLBindCol(hstmt, 3, SQL_C_CHAR, StatusArray, sizeof(StatusArray[0]),
StatusLenOrIndArray);

// Execute a statement to retrieve rows from the Orders table.
SQLExecDirect(hstmt, "SELECT OrderID, SalesPerson, Status FROM Orders", SQL_NTS);

// Fetch up to the rowset size number of rows at a time. Print the actual
// number of rows fetched; this number is returned in NumRowsFetched.
// Check the row status array to print only those rows successfully
// fetched. Code to check if rc equals SQL_SUCCESS_WITH_INFO or
// SQL_ERROR not shown.
while ((rc = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0)) != SQL_NO_DATA) {
    for (i = 0; i < NumRowsFetched; i++) {
        if ((RowStatusArray[i] == SQL_ROW_SUCCESS) ||
            (RowStatusArray[i] == SQL_ROW_SUCCESS_WITH_INFO)) {
            if (OrderIDIndArray[i] == SQL_NULL_DATA)
                printf("%s", OrderIDArray[i]);
            else
                printf("%d", OrderIDArray[i]);
            if (SalesPersonLenOrIndArray[i] == SQL_NULL_DATA)
                printf("%s" , SalesPersonArray[i]);
            else
                printf("%s", SalesPersonArray[i]);
            if (StatusLenOrIndArray[i] == SQL_NULL_DATA)
                printf("%s", StatusArray[i]);
            else
                printf("%s", StatusArray[i]);
        }
    }
}

// Close the cursor.
SQLCloseCursor(hstmt);
```

Row-Wise Binding
When using row-wise binding, an application defines a structure containing one or two, or in some cases three, elements for each column for which data is to be returned. The first element holds the data value, and the second element holds the length/indicator buffer. Indicators and length values can be stored in separate buffers by setting the SQL_DESC_INDICATOR_PTR and SQL_DESC_OCTET_LENGTH_PTR descriptor fields to different values; if this is done, the structure contains a third element. The application then allocates an array of these structures, which contains as many elements as there are rows in the rowset.

The application declares the size of the structure to the driver with the SQL_ATTR_ROW_BIND_TYPE statement attribute and binds the address of each member in the first element of the array. Thus, the driver can calculate the address of the data for a particular row and column as

\[
\text{Address} = \text{Bound Address} + ((\text{Row Number} - 1) \times \text{Structure Size})
\]

where rows are numbered from 1 to the size of the rowset. (One is subtracted from the row number because array indexing in C is zero-based.) The following illustration shows how row-wise binding works. Generally, only columns that will be bound are included in the structure. The structure can contain fields that are unrelated to result set columns. The columns can be placed in the structure in any order but are shown in sequential order for clarity.

For example, the following code creates a structure with elements in which to return data for the OrderID, SalesPerson, and Status columns, and length/indicators for the SalesPerson and Status columns. It allocates 10 of these structures and binds them to the OrderID, SalesPerson, and Status columns.

```c
#define ROW_ARRAY_SIZE 10

typedef struct {
    SQLUINTEGER OrderID;
    SQLINTEGER OrderIDInd;
    SQLCHAR SalesPerson[11];
    SQLINTEGER SalesPersonLenOrInd;
    SQLCHAR Status[7];
    SQLINTEGER StatusLenOrInd;
} ORDERINFO;

ORDERINFO OrderInfoArray[ROW_ARRAY_SIZE];

SQLULEN NumRowsFetched;
SQLUSMALLINT RowStatusArray[ROW_ARRAY_SIZE], i;
SQLRETURN rc;
SQLHSTMT hstmt;

// Specify the size of the structure with the SQL_ATTR_ROW_BIND_TYPE // statement attribute. This also declares that row-wise binding will // be used. Declare the rowset size with the SQL_ATTR_ROW_ARRAY_SIZE // statement attribute. Set the SQL_ATTR_ROW_STATUS_PTR statement // attribute to point to the row status array. Set the // SQL_ATTR_ROWS_FETCHED_PTR statement attribute to point to // NumRowsFetched.
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE, sizeof(ORDERINFO), 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, ROW_ARRAY_SIZE, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROWS_FETCHED_PTR, &NumRowsFetched, 0);

// Bind elements of the first structure in the array to the OrderID, // SalesPerson, and Status columns.
SQLBindCol(hstmt, 1, SQL_C_ULONG, &OrderInfoArray[0].OrderID, 0, &OrderInfoArray[0].OrderIDInd);
SQLBindCol(hstmt, 2, SQL_C_CHAR, OrderInfoArray[0].SalesPerson, sizeof(OrderInfoArray[0].SalesPerson), &OrderInfoArray[0].SalesPersonLenOrInd);
SQLBindCol(hstmt, 3, SQL_C_CHAR, OrderInfoArray[0].Status, sizeof(OrderInfoArray[0].Status), &OrderInfoArray[0].StatusLenOrInd);

// Execute a statement to retrieve rows from the Orders table.
SQLExecDirect(hstmt, "SELECT OrderID, SalesPerson, Status FROM Orders", SQL_NTS);

// Fetch up to the rowset size number of rows at a time. Print the actual // number of rows fetched; this number is returned in NumRowsFetched.
// Check the row status array to print only those rows successfully // fetched. Code to check if rc equals SQL_SUCCESS_WITH_INFO or // SQL_ERROR is not shown.
```
Using Block Cursors

Support for block cursors is built into ODBC 3.x. SQLFetch can be used only for multirow fetches when called in ODBC 3.x; if an ODBC 2.x application calls SQLFetch, it will open only a single-row, forward-only cursor. When an ODBC 3.x application calls SQLFetch in an ODBC 2.x driver, it returns a single row unless the driver supports SQLExtendedFetch. For more information, see Block Cursors, Scrollable Cursors, and Backward Compatibility in Appendix G: Driver Guidelines for Backward Compatibility.

To use block cursors, the application sets the rowset size, binds the rowset buffers (as described in the previous section), optionally sets the SQL_ATTR_ROWS_FETCHED_PTR and SQL_ATTR_ROW_STATUS_PTR statement attributes, and calls SQLFetch or SQLFetchScroll to fetch a block of rows. The application can change the rowset size and bind new rowset buffers (by calling SQLBindCol or specifying a bind offset) even after rows have been fetched.

This section contains the following topics.

- Rowset Size
- Number of Rows Fetched and Status
- SQLGetData and Block Cursors; block cursor

Rowset Size

Which rowset size to use depends on the application. Screen-based applications commonly follow one of two strategies. The first is to set the rowset size to the number of rows displayed on the screen; if the user resizes the screen, the application changes the rowset size accordingly. The second is to set the rowset size to a larger number, such as 100, which reduces the number of calls to the data source. The application scrolls locally within the rowset when possible and fetches new rows only when it scrolls outside the rowset.

Other applications, such as reports, tend to set the rowset size to the largest number of rows the application can reasonably handle — with a larger rowset, the network overhead per row is sometimes reduced. Exactly how large a rowset can be depends on the size of each row and the amount of memory available.

Rowset size is set by a call to SQLSetStmtAttr with an Attribute argument of SQL_ATTR_ROWS_ARRAY_SIZE. The application can change the rowset size and bind new rowset buffers (by calling SQLBindCol or specifying a bind offset) even after rows have been fetched, or both. The implications of changing the rowset size depend on the function:

- SQLFetch and SQLFetchScroll use the rowset size at the time of the call to determine how many rows to fetch. However, SQLFetchScroll with a FetchOrientation of SQL_FETCH_NEXT increments the cursor based on the rowset of the previous fetch and then fetches a rowset based on the current rowset size.
- SQLSetPos uses the rowset size that is in effect as of the preceding call to SQLFetch or SQLFetchScroll, because SQLSetPos operates on a rowset that has already been set. SQLSetPos also will pick up the new rowset size if SQLBulkOperations has been called after the rowset size was changed.
- SQLBulkOperations uses the rowset size in effect at the time of the call, because it performs operations on a table independent of any fetched rowset.
Number of Rows Fetched and Status

If the SQL_ATTR_ROWS_Fetched_PTR statement attribute has been set, it specifies a buffer that returns the number of rows fetched by the call to SQLFetch or SQLFetchScroll, and error rows. (This number is a count of all rows that do not have the status SQL_ROW_NO_ROWS.) After a call to SQLBulkOperations or SQLSetPos, the buffer contains the number of rows that were affected by a bulk operation performed by the function. If the SQL_ATTR_ROW_STATUS_PTR statement attribute has been set, SQLFetch or SQLFetchScroll returns the row status array, which provides the status of each returned row. Both of the buffers pointed to by these fields are allocated by the application and populated by the driver. An application must make sure that these pointers remain valid until the cursor is closed.

Entries in the row status array state whether each row was fetched successfully, whether it was updated, added, or deleted since it was last fetched, and whether an error occurred while fetching the row. If SQLFetch or SQLFetchScroll encounters an error while retrieving one row of a multirow rowset, or if SQLBulkOperations with an Operation argument of SQL_FETCH_BY_BOOKMARK encounters an error while performing a bulk fetch, it sets the corresponding value in the row status array to SQL_ROW_ERROR, continues fetching rows, and returns SQL_SUCCESS_WITH_INFO. For more information about error handling and the row status array, see the SQLFetch and SQLFetchScroll function descriptions.

© 2016 Microsoft

SQLGetData and Block Cursors

SQLGetData operates on a single column of a single row and cannot fetch an array containing data from multiple rows. This is because the primary use of SQLGetData is to fetch long data in parts, and there is little or no reason to do this for more than one row at a time.

To use SQLGetData with a block cursor, an application first calls SQLSetPos to position the cursor on a single row. It then calls SQLGetData for a column in that row. However, this behavior is optional. To determine if a driver supports the use of SQLGetData with block cursors, an application calls SQLGetInfo with the SQL_GETDATA_EXTENSIONS option.

© 2016 Microsoft

Row Status Array

In addition to data, SQLFetch and SQLFetchScroll can return an array that gives the status of each row in the rowset. This array is specified through the SQL_ATTR_ROW_STATUS_PTR statement attribute. This array is allocated by the application and must have as many elements as are specified by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. The values in the array are set by SQLBulkOperations, SQLFetch, SQLFetchScroll, and SQLSetPos. The values describe the status of the row and whether that status has changed since it was last fetched.

<table>
<thead>
<tr>
<th>Row status array value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ROW_SUCCESS</td>
<td>The row was successfully fetched and has not changed since it was last fetched.</td>
</tr>
<tr>
<td>SQL_ROW_SUCCESS_WITH_INFO</td>
<td>The row was successfully fetched and has not changed since it was last fetched. However, a warning was returned about the row.</td>
</tr>
<tr>
<td>SQL_ROW_ERROR</td>
<td>An error occurred while fetching the row.</td>
</tr>
<tr>
<td>SQL_ROW_UPDATED</td>
<td>The row was successfully fetched and has been updated since it was last fetched. If the row is fetched again or refreshed by SQLSetPos, its status is changed to the new status. Some drivers cannot detect changes to data and therefore cannot return this value. To determine whether a driver can detect updates to refetched rows, an application calls SQLGetInfo with the SQL_ROW_UPDATES option.</td>
</tr>
<tr>
<td>SQL_ROW_DELETED</td>
<td>The row has been deleted since it was last fetched.</td>
</tr>
<tr>
<td>SQL_ROW_ADDED</td>
<td>The row was inserted by SQLBulkOperations. If the row is fetched again or is refreshed by SQLSetPos, its status is SQL_ROW_SUCCESS. This value is not set by SQLFetch or SQLFetchScroll.</td>
</tr>
<tr>
<td>SQL_ROW_NOROW</td>
<td>The rowset overlapped the end of the result set, and no row was returned that corresponded to this element of the row status array.</td>
</tr>
</tbody>
</table>
Scrollable Cursors

In modern screen-based applications, the user scrolls backward and forward through the data. For such applications, returning to a previously fetched row is a problem. One possibility is to close and reopen the cursor and then fetch rows until the cursor reaches the required row. Another possibility is to read the result set, cache it locally, and implement scrolling in the application. Both possibilities work well only with small result sets, and the latter possibility is difficult to implement. A better solution is to use a scrollable cursor, which can move backward and forward in the result set.

A scrollable cursor is commonly used in modern screen-based applications in which the user scrolls back and forth through the data. However, applications should use scrollable cursors only when forward-only cursors will not do the job, as scrollable cursors are generally more expensive than forward-only cursors.

The ability to move backward raises a question not applicable to forward-only cursors: Should a scrollable cursor detect changes made to rows previously fetched? That is, should it detect updated, deleted, and newly inserted rows?

This question arises because the definition of a result set — the set of rows that matches certain criteria — does not state when rows are checked to see whether they match that criteria, nor does it state whether rows must contain the same data each time they are fetched. The former omission makes it possible for scrollable cursors to detect whether rows have been inserted or deleted, while the latter makes it possible for them to detect updated data.

The ability to detect changes is sometimes useful, sometimes not. For example, an accounting application needs a cursor that ignores all changes; balancing books is impossible if the cursor shows the latest changes. On the other hand, an airline reservation system needs a cursor that shows the latest changes to the data; without such a cursor, it must continually requery the database to show the most up-to-date flight availability.

To cover the needs of different applications, ODBC defines four different types of scrollable cursors. These cursors vary both in expense and in their ability to detect changes to the result set. Note that if a scrollable cursor can detect changes to rows, it can only detect them 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. Note as well that visibility of changes is also controlled by the transaction isolation level; for more information, see Transaction Isolation.

This section contains the following topics.

- Scrollable Cursor Types
- Using Scrollable Cursors
- Relative and Absolute Scrolling
- Bookmarks

© 2016 Microsoft

Scrollable Cursor Types

The four types of scrollable cursors are static, dynamic, keyset-driven, and mixed. Static cursors detect few or no changes but are relatively cheap to implement. Dynamic cursors detect all changes but are expensive to implement. Keyset-driven and mixed cursors lie in between, detecting most changes but at less expense than dynamic cursors.

The following terms are used to define the characteristics of each type of scrollable cursor:
- **Own updates, deletes, and inserts.** Updates, deletes, and inserts made through the cursor, either with a call to `SQLBulkOperations` or `SQLSetPos` or with a positioned update or delete statement.

- **Other updates, deletes, and inserts.** Updates, deletes, and inserts not made by the cursor, including those made by other operations in the same transaction, those made through other transactions, and those made by other applications.

- **Membership.** The set of rows in the result set.

- **Order.** The order in which rows are returned by the cursor.

- **Values.** The values in each row in the result set.

For information about how to update, delete, and insert data, see [Updating Data Overview](#).

This section contains the following topics.

- ODBC Static Cursors
- ODBC Dynamic Cursors
- Keyset-Driven Cursors
- Mixed Cursors

---

**ODBC Static Cursors**

A static cursor is one in which the result set appears to be static. It does not usually detect changes that were made to the membership, order, or values of the result set after the cursor is opened. For example, suppose a static cursor fetches a row and another application then updates that row. If the static cursor refetches the row, the values it sees are unchanged, despite the changes that were made by the other application.

Static cursors can detect their own updates, deletes, and inserts, although they are not required to do this. Whether a particular static cursor detects these changes is reported through the `SQL_STATICSENSITIVITY` option in `SQLGetInfo`. Static cursors never detect other updates, deletes, and inserts.

The row status array specified by the `SQL_ATTR_ROW_STATUS_PTR` statement attribute can contain `SQL_ROW_SUCCESS`, `SQL_ROW_SUCCESS_WITH_INFO`, or `SQL_ROW_ERROR` for any row. It returns `SQL_ROW_UPDATED`, `SQL_ROW_DELETED`, or `SQL_ROW_ADDED` for rows updated, deleted, or inserted by the cursor, assuming that the cursor can detect such changes.

Static cursors are typically implemented by locking the rows in the result set or by making a copy, or snapshot, of the result set. Although locking rows is relatively easy to do, it has the drawback of significantly reducing concurrency. Making a copy allows for greater concurrency and allows the cursor to keep track of its own updates, deletes, and inserts by modifying the copy. However, a copy is more expensive to make and can diverge from the underlying data as that data is changed by others.

---

**ODBC Dynamic Cursors**

A dynamic cursor is just that: dynamic. It can detect any changes made to the membership, order, and values of the
result set after the cursor is opened. For example, suppose a dynamic cursor fetches two rows and another application then updates one of those rows and deletes the other. If the dynamic cursor then attempts to refetch those rows, it will not find the deleted row but will return the new values for the updated row.

Dynamic cursors detect all updates, deletes, and inserts, both their own and those made by others. (This is subject to the isolation level of the transaction, as set by the SQL_ATTR_TXN_ISOLATION connection attribute.) The row status array specified by the SQL_ATTR_ROW_STATUS_PTR statement attribute reflects these changes and can contain SQL_ROW_SUCCESS, SQL_ROW_SUCCESS_WITH_INFO, SQL_ROW_ERROR, SQL_ROW_UPDATED, and SQL_ROW_ADDED. It cannot return SQL_ROW_DELETED because a dynamic cursor does not return deleted rows outside the rowset and therefore no longer recognizes the existence of the deleted row in the result set or its corresponding element in the row status array. SQL_ROW_ADDED is returned only when a row is updated by a call to SQLSetPos, not when it is updated by another cursor.

One way of implementing dynamic cursors in the database is by creating a selective index that defines the membership and ordering of the result set. Because the index is updated when others make changes, a cursor based on such an index is sensitive to all changes. Additional selection within the result set defined by this index is possible by processing along the index.

Dynamic cursors can be simulated by requiring the result set to be ordered by a unique key. With such a restriction, fetches are made by executing a SELECT statement each time the cursor fetches rows. For example, suppose the result set is defined by this statement:

```sql
SELECT * FROM Customers ORDER BY Name, CustID
```

To fetch the next rowset in this result set, the simulated cursor sets the parameters in the following SELECT statement to the values in the last row of the current rowset, and then executes it:

```sql
SELECT * FROM Customers WHERE (Name > ?) AND (CustID > ?)
ORDER BY Name, CustID
```

This statement creates a second result set, the first rowset of which is the next rowset in the original result set — in this case, the set of rows in the Customers table. The cursor returns this rowset to the application.

It is interesting to note that a dynamic cursor implemented in this manner actually creates many result sets, which allows it to detect changes to the original result set. The application never learns of the existence of these auxiliary result sets; it simply appears as if the cursor is able to detect changes to the original result set.

© 2016 Microsoft

Keyset-Driven Cursors

A keyset-driven cursor lies between a static and a dynamic cursor in its ability to detect changes. Like a static cursor, it does not always detect changes to the membership and order of the result set. Like a dynamic cursor, it does detect changes to the values of rows in the result set (subject to the isolation level of the transaction, as set by the SQL_ATTR_TXN_ISOLATION connection attribute).

When a keyset-driven cursor is opened, it saves the keys for the entire result set; this fixes the apparent membership and order of the result set. As the cursor scrolls through the result set, it uses the keys in this keyset to retrieve the current data values for each row. For example, suppose a keyset-driven cursor fetches a row and another application then updates that row. If the cursor refetches the row, the values it sees are the new ones because it refetched the row using its key. Because of this, the keyset-driven cursors always detect changes made by themselves and others.

When the cursor attempts to retrieve a row that has been deleted, this row appears as a "hole" in the result set: The
key for the row exists in the keyset, but the row no longer exists in the result set. If the key values in a row are updated, the row is considered to have been deleted and then inserted, so such rows also appear as holes in the result set. While a keyset-driven cursor can always detect rows deleted by others, it can optionally remove the keys for rows it deletes itself from the keyset. Keyset-driven cursors that do this cannot detect their own deletes. Whether a particular keyset-driven cursor detects its own deletes is reported through the SQL_STATICSENSITIVITY option in SQLGetInfo.

Rows inserted by others are never visible to a keyset-driven cursor because no keys for these rows exist in the keyset. However, a keyset-driven cursor can optionally add the keys for rows it inserts itself to the keyset. Keyset-driven cursors that do this can detect their own inserts. Whether a particular keyset-driven cursor detects its own inserts is reported through the SQL_STATICSENSITIVITY option in SQLGetInfo.

The row status array specified by the SQL_ATTR_ROW_STATUS_PTR statement attribute can contain SQL_ROW_SUCCESS, SQL_ROW_SUCCESS_WITH_INFO, or SQL_ROW_ERROR for any row. It returns SQL_ROW_UPDATED, SQL_ROW_DELETED, or SQL_ROW_ADDED for rows it detects as updated, deleted, or inserted.

Keyset-driven cursors are commonly implemented by creating a temporary table that contains the keys for each row in the result set. Because the cursor must also determine whether rows have been updated, this table also contains a column with row versioning information.

To scroll over the original result set, the keyset-driven cursor opens a static cursor over the temporary table. To retrieve a row in the original result set, the cursor first retrieves the appropriate key from the temporary table and then retrieves the current values for the row. If block cursors are used, the cursor must retrieve multiple keys and rows.

Mixed Cursors

A mixed cursor is a combination of a keyset-driven cursor and a dynamic cursor. It is used when the result set is too large to reasonably save keys for the entire result set. Mixed cursors are implemented by creating a keyset that is smaller than the entire result set but larger than the rowset.

As long as the application scrolls within the keyset, the behavior is keyset-driven. When the application scrolls outside the keyset, the behavior is dynamic: The cursor fetches the requested rows and creates a new keyset. After the new keyset is created, the behavior reverts to keyset-driven within that keyset.

For example, suppose a result set has 1,000 rows and uses a mixed cursor with a keyset size of 100 and a rowset size of 10. When the first rowset is fetched, the cursor creates a keyset consisting of the keys for the first 100 rows. It then returns the first 10 rows, as requested.

Now suppose another application deletes rows 11 and 101. If the cursor attempts to retrieve row 11, it will encounter a hole because it has a key for this row but no row exists; this is keyset-driven behavior. If the cursor attempts to retrieve row 101, the cursor will not detect that the row is missing because it does not have a key for the row. Instead, it will retrieve what was previously row 102. This is dynamic cursor behavior.

A mixed cursor is equivalent to a keyset-driven cursor when the keyset size is equal to the result set size. A mixed cursor is equivalent to a dynamic cursor when the keyset size is equal to 1.
1. Determine the cursor capabilities.
2. Set up the cursor.
3. Scroll and fetch rows.

This section contains the following topics.

- Determining Cursor Capabilities
- Setting Up the Cursor
- Cursor Characteristics and Cursor Type
- Scrolling and Fetching Rows

Determining Cursor Capabilities

The following four options in `SQLGetInfo` describe what types of cursors are supported and what their capabilities are:

- SQL_CURSOR_SENSITIVITY. Indicates whether a cursor is sensitive to changes made by another cursor.
- SQL_SCROLL_OPTIONS. Lists the supported cursor types (forward-only, static, keyset-driven, dynamic, or mixed). All data sources must support forward-only cursors.
- SQL_DYNAMIC_CURSOR_ATTRIBUTES1, SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1, SQL_KEYSET_CURSOR_ATTRIBUTES1, or SQL_STATIC_CURSOR_ATTRIBUTES1 (depending on the type of the cursor). Lists the fetch types supported by scrollable cursors. The bits in the return value correspond to the fetch types in `SQLFetchScroll`.
- SQL_KEYSET_CURSOR_ATTRIBUTES2 or SQL_STATIC_CURSOR_ATTRIBUTES2 (depending on the type of the cursor). Lists whether static and keyset-driven cursors can detect their own updates, deletes, and inserts.

An application can determine cursor capabilities at run time by calling `SQLGetInfo` with these options. This is commonly done by generic applications. Cursor capabilities also can be determined during application development and their use hard-coded into the application. This is commonly done by vertical and custom applications but can also be done by generic applications that use a client-side cursor implementation such as the ODBC cursor library.

Setting Up the Cursor

The application can specify the cursor type before executing a statement that creates a result set. It does this with the SQL_ATTR_CURSOR_TYPE statement attribute. If the application does not explicitly specify a type, a forward-only cursor will be used. To get a mixed cursor, an application specifies a keyset-driven cursor but declares a keyset size less than the result set size.

For keyset-driven and mixed cursors, the application can also specify the keyset size. It does this with the SQL_ATTR_KEYSET_SIZE statement attribute. If the keyset size is set to 0, which is the default, the keyset size is set to the result set size and a keyset-driven cursor is used. The keyset size can be changed after the cursor has been opened.
The application can also set the rowset size; for more information, see Using Block Cursors, earlier in this section.

Cursor Characteristics and Cursor Type

An application can specify the characteristics of a cursor instead of specifying the cursor type (forward-only, static, keyset-driven, or dynamic). To do this, the application selects the cursor's scrollability (by setting the SQL_ATTR_CURSOR_SCROLLABLE statement attribute) and sensitivity (by setting the SQL_ATTR_CURSOR_SENSITIVITY statement attribute) before opening the cursor on the statement handle. The driver then chooses the cursor type that most efficiently provides the characteristics that the application requested.

Whenever an application sets any of the statement attributes SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_SCROLLABLE, SQL_ATTR_CURSOR_SENSITIVITY, or SQL_ATTR_CURSOR_TYPE, the driver makes any required change to the other statement attributes in this set of four attributes so that their values remain consistent. As a result, when the application specifies a cursor characteristic, the driver can change the attribute that indicates cursor type based on this implicit selection; when the application specifies a type, the driver can change any of the other attributes to be consistent with the characteristics of the selected type. For more information about these statement attributes, see the SQLSetStmtAttr function description.

An application that sets statement attributes to specify both a cursor type and cursor characteristics runs the risk of obtaining a cursor that is not the most efficient method available on that driver of meeting the application's requirements.

The implicit setting of statement attributes is driver-defined except that it must follow these rules:

- Forward-only cursors are never scrollable; see the definition of SQL_ATTR_CURSOR_SCROLLABLE in SQLSetStmtAttr.
- Insensitive cursors are never updatable (and thus their concurrency is read-only); this is based on their definition of insensitive cursors in the ISO SQL standard.

Consequently, the implicit setting of statement attributes occurs in the cases described in the following table.

<table>
<thead>
<tr>
<th>Application sets attribute to</th>
<th>Other attributes set implicitly</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_CONCURRENCY to SQL_CONCUR_READ_ONLY</td>
<td>SQL_ATTR_CURSOR_SENSITIVITY to SQL_INSENSITIVE.</td>
</tr>
<tr>
<td>SQL_ATTR_CONCURRENCY to SQL_CONCUR_LOCK,</td>
<td>SQL_ATTR_CURSOR_SENSITIVITY to SQL_UNSPECIFIED or SQL_SENSITIVE,</td>
</tr>
<tr>
<td>SQL_CONCUR_ROWVER, or</td>
<td>as defined by the driver. It can never be set to SQL_INSENSITIVE,</td>
</tr>
<tr>
<td>SQL_CONCUR_VALUES</td>
<td>because insensitive cursors are always read-only.</td>
</tr>
<tr>
<td>SQL_ATTR_CURSOR_SCROLLABLE to SQL_NONSCROLLABLE</td>
<td>SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_FORWARD_ONLY</td>
</tr>
<tr>
<td>SQL_ATTR_CURSOR_SCROLLABLE to SQL_SCROLLABLE</td>
<td>SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_STATIC,</td>
</tr>
<tr>
<td></td>
<td>SQL_CURSOR_KEYSET_DRIVEN, or SQL_CURSOR_DYNAMIC, as specified by</td>
</tr>
<tr>
<td></td>
<td>the driver. It is never set to SQL_CURSOR_FORWARD_ONLY.</td>
</tr>
<tr>
<td>SQL_ATTR_CURSOR_SENSITIVITY to SQL_INSENSITIVE</td>
<td>SQL_ATTR_CONCURRENCY to SQL_CONCUR_READ_ONLY.</td>
</tr>
<tr>
<td></td>
<td>SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_STATIC.</td>
</tr>
</tbody>
</table>
| SQL_ATTR_CURSOR_SENSITIVITY to SQL_SENSITIVE | SQL_ATTR_CONCURRENCY to SQL_CONCUR_LOCK, SQL_CONCUR_ROWVER, or SQL_CONCUR_VALUES, as specified by the driver. It is never set to SQL_CONCUR_READ_ONLY.
SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_FORWARD_ONLY, SQL_CURSOR_STATIC, SQL_CURSOR_KEYSET_DRIVEN, or SQL_CURSOR_DYNAMIC, as specified by the driver. |
| SQL_ATTR_CURSOR_SENSITIVITY to SQL_UNSPECIFIED | SQL_ATTR_CONCURRENCY to SQL_CONCUR_READ_ONLY, SQL_CONCUR_LOCK, SQL_CONCUR_ROWVER, or SQL_CONCUR_VALUES, as specified by the driver.
SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_FORWARD_ONLY, SQL_CURSOR_STATIC, SQL_CURSOR_KEYSET_DRIVEN, or SQL_CURSOR_DYNAMIC, as specified by the driver. |
| SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_DYNAMIC | SQL_ATTR_SCROLLABLE to SQL_SCROLLABLE.
SQL_ATTR_CURSOR_SENSITIVITY to SQL_SENSITIVE. (But only if SQL_ATTR_CONCURRENCY is not equal to SQL_CONCUR_READ_ONLY. Updatable dynamic cursors are always sensitive to changes that were made in their own transaction.) |
| SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_FORWARD_ONLY | SQL_ATTR_CURSOR_SCROLLABLE to SQL_NONSCROLLABLE. |
| SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_KEYSET_DRIVEN | SQL_ATTR_SCROLLABLE to SQL_SCROLLABLE.
SQL_ATTR_SENSITIVITY to SQL_UNSPECIFIED or SQL_SENSITIVE (according to driver-defined criteria, if SQL_ATTR_CONCURRENCY is not SQL_CONCUR_READ_ONLY). |
| SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_STATIC | SQL_ATTR_SCROLLABLE to SQL_SCROLLABLE.
SQL_ATTR_SENSITIVITY to SQL_INSENSITIVE (if SQL_ATTR_CONCURRENCY is SQL_CONCUR_READ_ONLY).
SQL_ATTR_SENSITIVITY to SQL_UNSPECIFIED or SQL_SENSITIVE (if SQL_ATTR_CONCURRENCY is not SQL_CONCUR_READ_ONLY). |

© 2016 Microsoft

Scrolling and Fetching Rows (ODBC)

When using a scrollable cursor, applications call `SQLFetchScroll` to position the cursor and fetch rows. `SQLFetchScroll` supports relative scrolling (next, prior, and relative n rows), absolute scrolling (first, last, and row n), and positioning by bookmark. The `FetchOrientation` and `FetchOffset` arguments in `SQLFetchScroll` specify which rowset to fetch, as shown in the following diagrams.
SQLFetchScroll positions the cursor to the specified row and returns the rows in the rowset starting with that row. If the specified rowset overlaps the end of the result set, a partial rowset is returned. If the specified rowset overlaps the start of the result set, the first rowset in the result set is usually returned; for complete details, see the SQLFetchScroll function description.

In some cases, the application might want to position the cursor without retrieving any data. For example, it might want to test whether a row exists or just get the bookmark for the row without bringing other data across the network. To do this, it sets the SQL_ATTR_RETRIEVE_DATA statement attribute to SQL_RD_OFF. The variable bound to the bookmark column (if any) is always updated, regardless of the setting of this statement attribute.

After the rowset has been retrieved, the application can call SQLSetPos to position to a particular row in the rowset or refresh rows in the rowset. For more information on using SQLSetPos, see Updating Data with SQLSetPos.

Note

Scrolling is supported in ODBC 2.x drivers by SQLExtendedFetch. For more information, see Block Cursors, Scrollable Cursors, and Backward Compatibility in Appendix G: Driver Guidelines for Backward Compatibility.
Relative and Absolute Scrolling

Most of the scrolling options in SQLFetchScroll position the cursor relative to the current position or to an absolute position. 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.

Dynamic cursors detect rows inserted into and deleted from the result set, so there is no easy way for dynamic cursors to retrieve the row at a particular number other than reading from the start of the result set, which is likely to be slow. Furthermore, absolute fetching is not very useful in dynamic cursors because row numbers change as rows are inserted and deleted; therefore, successively fetching the same row number can yield different rows.

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 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.

<table>
<thead>
<tr>
<th>Scroll bar operation</th>
<th>SQLFetchScroll scrolling option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page up</td>
<td>SQL_FETCH_PRIOR</td>
</tr>
<tr>
<td>Page down</td>
<td>SQL_FETCH_NEXT</td>
</tr>
<tr>
<td>Line up</td>
<td>SQL_FETCH_RELATIVE with FetchOffset equal to (-1)</td>
</tr>
<tr>
<td>Line down</td>
<td>SQL_FETCH_RELATIVE with FetchOffset equal to (1)</td>
</tr>
<tr>
<td>Scroll box at top</td>
<td>SQL_FETCH_FIRST</td>
</tr>
<tr>
<td>Scroll box at bottom</td>
<td>SQL_FETCH_LAST</td>
</tr>
<tr>
<td>Random scroll box position</td>
<td>SQL_FETCH_ABSOLUTE</td>
</tr>
</tbody>
</table>

Such applications also need to position the scroll box after a scrolling operation, which requires the current row number and the number of rows. For the current row number, applications can either keep track of the current row number or call SQLGetStmtAttr with the SQL_ATTR_ROW_NUMBER attribute to retrieve it.

The number of rows in the cursor, which is the size of the result set, is available as the SQL_DIAG_CURSOR_ROW_COUNT field of the diagnostic header. The value in this field is defined only after SQLExecute, SQLExecDirect, or SQLMoreResult has been called. This count can be either an approximate count or an exact count, depending on the capabilities of the driver. The driver’s support can be determined by calling SQLGetInfo with the cursor attributes information types and checking whether the SQL_CA2_CRC_APPROXIMATE or SQL_CA2_CRC_EXACT bit is returned for the type of cursor.

An exact row count is never supported for a dynamic cursor. For other types of cursors, the driver can support either exact or approximate row counts, but not both. If the driver supports neither exact nor approximate row counts for a specific cursor type, the SQL_DIAG_CURSOR_ROW_COUNT field contains the number of rows that have been fetched so far. Regardless of what the driver supports, SQLFetchScroll with an Operation of SQL_FETCH_LAST will cause the SQL_DIAG_CURSOR_ROW_COUNT field to contain the exact row count.
Bookmarks (ODBC)

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. Bookmarks in ODBC are a bit different from bookmarks in real books. In a real book, the reader places a bookmark at a specific page and then looks for that bookmark to return to the page. 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. Thus, bookmarks in ODBC are similar to a reader writing down a page number, remembering it, and then looking up the page again.

To determine a driver's support of bookmarks, an application calls SQLGetInfo with the SQL_BOOKMARK_PERSISTENCE option. The bits in this value describe what operations bookmarks survive, such as whether bookmarks are still valid after the cursor is closed.

This section contains the following topics.

- Bookmark Types
- Retrieving Bookmarks
- Scrolling by Bookmark
- Updating, Deleting, or Fetching by Bookmark
- Comparing Bookmarks

© 2016 Microsoft

Bookmark Types

All bookmarks in ODBC 3.x are variable-length bookmarks. This allows a primary key or a unique index associated with a table to be used as a bookmark. The bookmark also can be a 32-bit value, as was used in ODBC 2.x. To specify that a bookmark is used with a cursor, an ODBC 3.x application sets the SQL_ATTR_USE_BOOKMARK statement attribute to SQL_UB_VARIABLE. A variable-length bookmark is automatically used.

An application can call SQLColAttribute with the FieldIdentifier argument set to SQL_DESC_OCTET_LENGTH to obtain the length of the bookmark. Because a variable-length bookmark can be a long value, an application should not bind to column 0 unless it will use the bookmark for many of the rows in the rowset.

Fixed-length bookmarks are supported only for backward compatibility. If an ODBC 2.x application working with an ODBC 3.x driver calls SQLSetStmtOption to set SQL_USE_BOOKMARKS to SQL_UB_ON, it is mapped in the Driver Manager to SQL_UB_VARIABLE. A variable-length bookmark is used, even if only 32 bits of it are populated. If a driver supports fixed-length bookmarks, it will support variable-length bookmarks. If an ODBC 3.x application working with an ODBC 2.x driver calls SQLSetStmtAttr to set SQL_ATTR_USE_BOOKMARKS to SQL_UB_VARIABLE, it is mapped in the Driver Manager to SQL_UB_VARIABLE and a 32-bit fixed-length bookmark is used. The SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute must then point to a 32-bit bookmark. If the bookmarks used are longer than 32 bits, such as when primary keys are used as bookmarks, the cursor must map the actual values to 32-bit values. It could, for example, build a hash table of them. When an ODBC 3.x application working with an ODBC 2.x driver binds a bookmark, the buffer length must be 4.

© 2016 Microsoft

Retrieving Bookmarks
If the application will use bookmarks, it must set the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE before preparing or executing the statement. This is necessary because building and maintaining bookmarks can be an expensive operation, so bookmarks should be enabled only when an application can make good use of them.

Bookmarks are returned as column 0 of the result set. There are three ways an application can retrieve them:

- Bind column 0 of the result set. SQLFetch or SQLFetchScroll returns the bookmarks for each row in the rowset along with the data for other bound columns.
- Call SQLSetPos to position to a row in the rowset and then call SQLGetData for column 0. If a driver supports bookmarks, it must always support the ability to call SQLGetData for column 0, even if it does not allow applications to call SQLGetData for other columns before the last bound column.
- Call SQLBulkOperations with the Operation argument set to SQL_ADD, and column 0 bound. The cursor inserts the row and returns the bookmark for the row in the bound buffer.

© 2016 Microsoft

Scrolling by Bookmark

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 statement 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. This use of the FetchOffset argument is not supported when used with ODBC 2.x drivers; when an application calls SQLFetchScroll in an ODBC 2.x driver with FetchOrientation set to SQL_FETCH_BOOKMARK, the FetchOffset argument must be set to 0.

© 2016 Microsoft

Updating, Deleting, or Fetching by Bookmark

Bookmarks can be used to identify data to be updated in the result set, deleted from the result set, or fetched from the result set to the rowset buffers. These operations are performed by a call to SQLBulkOperations with an Option argument of SQL_UPDATE_BY_BOOKMARK, SQL_DELETE_BY_BOOKMARK, or SQL_FETCH_BY_BOOKMARK. The bookmarks used in these operations are stored in column 0 of the rowset buffers. When updating by bookmark, the data that result set columns are updated to is retrieved from the rowset buffers. For more information, see Updating Data with SQLBulkOperations.

© 2016 Microsoft

Comparing Bookmarks
Because bookmarks are byte-comparable, they can be compared for equality or inequality. To do so, an application treats each bookmark as an array of bytes and compares two bookmarks byte-by-byte. Because bookmarks are guaranteed to be distinct only within a result set, it makes no sense to compare bookmarks that were obtained from different result sets.

The ODBC Cursor Library

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

Block and scrollable cursors are very useful additions to many applications. However, not all drivers support block and scrollable cursors. The same is true of positioned update and delete statements and `SQLSetPos`, which are discussed in Updating Data. Therefore, the ODBC component of the Windows SDK, formerly included in the Microsoft Data Access Components (MDAC) SDK, includes a cursor library. The cursor library implements block, static cursors, positioned update and delete statements, and `SQLSetPos` for any driver that meets the Open Group Standard CLI conformance level. The cursor library may be redistributed with ODBC applications; see the licensing agreement in the SDK for more information.

To use the cursor library, an application sets the `SQL_ATTR_ODBC_CURSORS` connection attribute before it connects to the data source. For more information about the cursor library, see Appendix F: ODBC Cursor Library.

Multiple Results

A *result* is something returned by the data source after a statement is executed. ODBC has two types of results: result sets and row counts. *Row counts* are the number of rows affected by an update, delete, or insert statement. Batches, described in Batches of SQL Statements, can generate multiple results.

The following table lists the `SQLGetInfo` options an application uses to determine whether a data source returns multiple results for each different type of batch. In particular, a data source can return a single row count for the entire batch of statements or individual row counts for each statement in the batch. In the case of a result set–generating statement executed with an array of parameters, the data source can return a single result set for all sets of parameters or individual result sets for each set of parameters.
Procedure SQL_BATCH_ROW_COUNT[a] --[b]

Arrays of parameters SQL_PARAM_ARRAYS_ROW_COUNTS SQL_PARAM_ARRAYS_SELECTS

[a] Row count–generating statements in a batch may be supported, yet the return of the row counts not supported. The SQL_BATCH_SUPPORT option in SQLGetInfo indicates whether row count–generating statements are allowed in batches; the SQL_BATCH_ROW_COUNTS option indicates whether these row counts are returned to the application.

[b] Explicit batches and procedures always return multiple result sets when they include multiple result set–generating statements.

Note

The SQL_MULT_RESULT_SETS option introduced in ODBC 1.0 provides only general information about whether multiple result sets can be returned. In particular, it is set to "Y" if the SQL_BS_SELECT_EXPLICIT or SQL_BS_SELECT_PROC bits are returned for SQL_BATCH_SUPPORT or if SQL_PAS_BATCH is returned for SQL_PARAM_ARRAYS_SELECT.

To process multiple results, an application calls SQLMoreResults. This function discards the current result and makes the next result available. It returns SQL_NO_DATA when no more results are available. For example, suppose the following statements are executed as a batch:

SELECT * FROM Parts WHERE Price > 100.00;
UPDATE Parts SET Price = 0.9 * Price WHERE Price > 100.00

After these statements are executed, the application fetches rows from the result set created by the SELECT statement. When it is done fetching rows, it calls SQLMoreResults to make available the number of parts that were repriced. If necessary, SQLMoreResults discards unfetched rows and closes the cursor. The application then calls SQLRowCount to determine how many parts were repriced by the UPDATE statement.

It is driver-specific whether the entire batch statement is executed before any results are available. In some implementations, this is the case; in others, calling SQLMoreResults triggers the execution of the next statement in the batch.

If one of the statements in a batch fails, SQLMoreResults will return either SQL_ERROR or SQL_SUCCESS_WITH_INFO. If the batch was aborted when the statement failed or the failed statement was the last statement in the batch, SQLMoreResults will return SQL_ERROR. If the batch was not aborted when the statement failed and the failed statement was not the last statement in the batch, SQLMoreResults will return SQL_SUCCESS_WITH_INFO. SQL_SUCCESS_WITH_INFO indicates that at least one result set or count was generated and that the batch was not aborted.

© 2016 Microsoft

Updating Data Overview

Applications can update data either by executing SQL statements or by calling SQLSetPos or SQLBulkOperations. UPDATE, DELETE, and INSERT statements act directly on the data source and are usually supported by drivers. Searched update and delete statements contain a specification of the rows to change. Positioned update and delete statements and SQLSetPos act on the data source through a cursor and are less widely supported. Whether cursors can detect changes made to the result set with the methods described in this section depends on the type of the cursor and how it is implemented. Forward-only cursors do not revisit rows and therefore will not detect any changes. For information about whether scrollable cursors can detect changes, see Scrollable Cursors.

This section contains the following topics.
UPDATE, DELETE, and INSERT Statements

SQL-based applications make changes to tables by executing the UPDATE, DELETE, and INSERT statements. These statements are part of the Minimum SQL grammar conformance level and must be supported by all drivers and data sources.

The syntax of these statements is:

UPDATE table-name

    SET column-identifier = {expression | NULL}
    [, column-identifier = {expression | NULL}]...

    [WHERE search-condition]

DELETE FROM table-name[WHERE search-condition]

INSERT INTO table-name[(column-identifier [, column-identifier]...)]

    {query-specification | VALUES (insert-value [, insert-value]...)}

Note that the query-specification element is valid only in the Core and Extended SQL grammars, and that the expression and search-condition elements become more complex in the Core and Extended SQL grammars.

Like other SQL statements, UPDATE, DELETE, and INSERT statements are often more efficient when they use parameters. For example, the following statement can be prepared and repeatedly executed to insert multiple rows in the Orders table:

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

This efficiency can be increased by passing arrays of parameter values. For more information about statement parameters and arrays of parameter values, see Statement Parameters.
Applications can update or delete the current row in a result set with a positioned update or delete statement. Positioned update and delete statements are supported by some data sources, but not all of them. To determine whether a data source supports positioned update and delete statements, an application calls `SQLGetInfo` with the `SQL_DYNAMIC_CURSOR_ATTRIBUTES1`, `SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1`, `SQL_KEYSET_CURSOR_ATTRIBUTES1`, or `SQL_STATIC_CURSOR_ATTRIBUTES1` `InfoType` (depending on the type of the cursor). Note that the ODBC cursor library simulates positioned update and delete statements.

To use a positioned update or delete statement, the application must create a result set with a `SELECT FOR UPDATE` statement. The syntax of this statement is:

```
SELECT [ALL | DISTINCT] select-list
    FROM table-reference-list
    [WHERE search-condition]
    FOR UPDATE OF [column-name [, column-name]...]
```

The application then positions the cursor on the row to be updated or deleted. It can do this by calling `SQLFetchScroll` to retrieve a rowset containing the required row and calling `SQLSetPos` to position the rowset cursor on that row. The application then executes the positioned update or delete statement on a different statement than the statement being used by the result set. The syntax of these statements is:

```
UPDATE table-name
    SET column-identifier = {expression | NULL}
    [, column-identifier = {expression | NULL}]
    WHERE CURRENT OF cursor-name
```

`DELETE FROM table-name WHERE CURRENT OF cursor-name`

Notice that these statements require a cursor name. The application either can specify a cursor name with `SQLSetCursorName` before executing the statement that creates the result set or can let the data source automatically generate a cursor name when the cursor is created. In the latter case, the application retrieves this cursor name for use in positioned update and delete statements by calling `SQLGetCursorName`.

For example, the following code allows a user to scroll through the Customers table and delete customer records or update their addresses and phone numbers. It calls `SQLSetCursorName` to specify a cursor name before it creates the result set of customers and uses three statement handles: `hstmtCust` for the result set, `hstmtUpdate` for a positioned update statement, and `hstmtDelete` for a positioned delete statement. Although the code could bind separate variables to the parameters in the positioned update statement, it updates the rowset buffers and binds the elements of these buffers. This keeps the rowset buffers synchronized with the updated data.

```c
#define POSITIONED_UPDATE 100
#define POSITIONED_DELETE 101

SQLUINTEGER CustIDArray[10];
SQLCHAR NameArray[10][51], AddressArray[10][51], PhoneArray[10][11];
SQLINTEGER CustIDIndArray[10], NameLenOrIndArray[10], AddressLenOrIndArray[10], PhoneLenOrIndArray[10];
SQLUSMALLINT RowStatusArray[10], Action, RowNum;
SQLHSTMT hstmtCust, hstmtUpdate, hstmtDelete;

// Set the SQL_ATTR_BIND_TYPE statement attribute to use column-wise binding. Declare the rowset size with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Set the SQL_ATTR ROW_STATUS_PTR statement attribute to point to the row status array.
SQLSetStmtAttr(hstmtCust, SQL_ATTR_ROW_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
```
SQLSetStmtAttr(hstmtCust, SQL_ATTR_ROW_ARRAY_SIZE, 10, 0);
SQLSetStmtAttr(hstmtCust, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);

// Bind arrays to the CustID, Name, Address, and Phone columns.
SQLBindCol(hstmtCust, 1, SQL_C_ULONG, CustIDArray, 0, CustIDIndArray);
SQLBindCol(hstmtCust, 2, SQL_C_CHAR, NameArray, sizeof(NameArray[0]),
            NameLenOrIndArray);
SQLBindCol(hstmtCust, 3, SQL_C_CHAR, AddressArray, sizeof(AddressArray[0]),
            AddressLenOrIndArray);
SQLBindCol(hstmtCust, 4, SQL_C_CHAR, PhoneArray, sizeof(PhoneArray[0]),
            PhoneLenOrIndArray);

// Set the cursor name to Cust.
SQLSetCursorName(hstmtCust, "Cust", SQL_NTS);

// Prepare positioned update and delete statements.
SQLPrepare(hstmtUpdate, 
"UPDATE Customers SET Address = ?, Phone = ? WHERE CURRENT OF Cust",
SQL_NTS);
SQLPrepare(hstmtDelete, "DELETE FROM Customers WHERE CURRENT OF Cust", SQL_NTS);

// Execute a statement to retrieve rows from the Customers table.
SQLExecDirect(hstmtCust, 
"SELECT CustID, Name, Address, Phone FROM Customers FOR UPDATE OF Address, Phone",
SQL_NTS);

// Fetch and display the first 10 rows.
SQLFetchScroll(hstmtCust, SQL_FETCH_NEXT, 0);
DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray,
            AddressArray, AddressLenOrIndArray, PhoneArray, PhoneLenOrIndArray,
            RowStatusArray);

// Call GetAction to get an action and a row number from the user.
while (GetAction(&Action, &RowNum)) {
    switch (Action) {
    case SQL_FETCH_NEXT:
    case SQL_FETCH_PRIOR:
    case SQL_FETCH_FIRST:
    case SQL_FETCH_LAST:
    case SQL_FETCH_ABSOLUTE:
    case SQL_FETCH_RELATIVE:
        // Fetch and display the requested data.
        SQLFetchScroll(hstmtCust, Action, RowNum);
        DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray,
                    AddressArray, AddressLenOrIndArray, PhoneArray, PhoneLenOrIndArray,
                    RowStatusArray);
        break;

    case POSITIONED_UPDATE:
        // Get the new data and place it in the rowset buffers.
        GetNewData(AddressArray[RowNum - 1], &AddressLenOrIndArray[RowNum - 1],
                    PhoneArray[RowNum - 1], &PhoneLenOrIndArray[RowNum - 1]);

        // Bind the elements of the arrays at position RowNum-1 to the
        // parameters of the positioned update statement.
        SQLBindParameter(hstmtUpdate, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          50, 0, AddressArray[RowNum - 1], sizeof(AddressArray[0]),
                          &AddressLenOrIndArray[RowNum - 1]);
        SQLBindParameter(hstmtUpdate, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          10, 0, PhoneArray[RowNum - 1], sizeof(PhoneArray[0]),
                          &PhoneLenOrIndArray[RowNum - 1]);

        // Position the rowset cursor. The rowset is 1-based.
        SQLSetPos(hstmtCust, RowNum, SQL_POSITION, SQL_LOCK_NO_CHANGE);
    break;
    }
}
Simulating Positioned Update and Delete Statements

If the data source does not support positioned update and delete statements, the driver can simulate these. For example, the ODBC cursor library simulates positioned update and delete statements. The general strategy for simulating positioned update and delete statements is to convert positioned statements to searched ones. This is done by replacing the `WHERE CURRENT OF` clause with a searched `WHERE` clause that identifies the current row.

For example, because the CustID column uniquely identifies each row in the Customers table, the positioned delete statement

```sql
DELETE FROM Customers WHERE CURRENT OF CustCursor
```

might be converted to

```sql
DELETE FROM Customers WHERE (CustID = ?)
```

The driver may use one of the following row identifiers in the `WHERE` clause:

- Columns whose values serve to identify uniquely every row in the table. For example, calling `SQLSpecialColumns` with `SQL_BEST_ROWID` returns the optimal column or set of columns that serve this purpose.
- Pseudo-columns, provided by some data sources, for the purpose of uniquely identifying every row. These may also be retrievable by calling `SQLSpecialColumns`.
- A unique index, if available.
- All the columns in the result set.

Exactly which columns a driver should use in the `WHERE` clause it constructs depends on the driver. On some data sources, determining a row identifier can be costly. However, it is faster to execute and guarantees that a simulated statement updates or deletes at most one row. Depending on the capabilities of the underlying DBMS, using a row...
identifier can be expensive to set up. However, it is faster to execute and guarantees that a simulated statement will update or delete only one row. The option of using all the columns in the result set is usually much easier to set up. However, it is slower to execute and, if the columns do not uniquely identify a row, can result in rows being unintentionally updated or deleted, especially when the select list for the result set does not contain all the columns that exist in the underlying table.

Depending upon which of the preceding strategies the driver supports, an application can choose which strategy it wants the driver to use with the SQL_ATTR_SIMULATE_CURSOR statement attribute. Although it might seem odd for an application to risk unintentionally updating or deleting a row, the application can remove this risk by ensuring that the columns in the result set uniquely identify each row in the result set. This saves the driver the effort of having to do this.

If the driver chooses to use a row identifier, it intercepts the `SELECT FOR UPDATE` statement that creates the result set. If the columns in the select list do not effectively identify a row, the driver adds the necessary columns to the end of the select list. Some data sources have a single column that always uniquely identifies a row, such as the ROWID column in Oracle; if such a column is available, the driver uses this. Otherwise, the driver calls `SQLSpecialColumns` for each table in the `FROM` clause to retrieve a list of the columns that uniquely identify each row. A common restriction that results from this technique is that cursor simulation fails if there is more than one table in the `FROM` clause.

No matter how the driver identifies rows, it usually strips the `FOR UPDATE OF clause off the `SELECT FOR UPDATE` statement before sending it to the data source. The `FOR UPDATE OF` clause is used only with positioned update and delete statements. Data sources that do not support positioned update and delete statements generally do not support it.

When the application submits a positioned update or delete statement for execution, the driver replaces the `WHERE CURRENT OF` clause with a `WHERE` clause containing the row identifier. The values of these columns are retrieved from a cache maintained by the driver for each column it uses in the `WHERE` clause. After the driver has replaced the `WHERE` clause, it sends the statement to the data source for execution.

For example, suppose that the application submits the following statement to create a result set:

```sql
SELECT Name, Address, Phone FROM Customers FOR UPDATE OF Phone, Address
```

If the application has set SQL_ATTR_SIMULATE_CURSOR to request a guarantee of uniqueness and if the data source does not provide a pseudo-column that always uniquely identifies a row, the driver calls `SQLSpecialColumns` for the Customers table, discovers that CustID is the key to the Customers table and adds this to the select list, and strips the `FOR UPDATE OF` clause:

```sql
SELECT Name, Address, Phone, CustID FROM Customers
```

If the application has not requested a guarantee of uniqueness, the driver strips only the `FOR UPDATE OF` clause:

```sql
SELECT Name, Address, Phone FROM Customers
```

Suppose the application scrolls through the result set and submits the following positioned update statement for execution, where Cust is the name of the cursor over the result set:

```sql
UPDATE Customers SET Address = ?, Phone = ? WHERE CURRENT OF Cust
```

If the application has not requested a guarantee of uniqueness, the driver replaces the `WHERE` clause and binds the CustID parameter to the variable in its cache:
If the application has not requested a guarantee of uniqueness, the driver replaces the **WHERE** clause and binds the Name, Address, and Phone parameters in this clause to the variables in its cache:

```sql
UPDATE Customers SET Address = ?, Phone = ?
    WHERE (Name = ?) AND (Address = ?) AND (Phone = ?)
```

### Determining the Number of Affected Rows

After an application updates, deletes, or inserts rows, it can call **SQLRowCount** to determine how many rows were affected. **SQLRowCount** returns this value whether or not the rows were updated, deleted, or inserted by executing an **UPDATE**, **DELETE**, or **INSERT** statement, by executing a positioned update or delete statement, or by calling **SQLSetPos**.

If a batch of SQL statements is executed, the count of affected rows might be a total count for all statements in the batch or individual counts for each statement in the batch. For more information, see [Batches of SQL Statements](#) and [Multiple Results](#).

The number of affected rows is also returned in the SQL_DIAG_ROW_COUNT diagnostic header field in the diagnostic area associated with the statement handle. However, the data in this field is reset after every function call on the same statement handle, whereas the value returned by **SQLRowCount** remains the same until a call to **SQLBulkOperations**, **SQLExecute**, **SQLExecDirect**, **SQLPrepare**, or **SQLSetPos**.

### Updating Data with **SQLSetPos**

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. It lets an ODBC driver support positioned updates even when the data source does not support positioned SQL statements. It is part of the paradigm of achieving complete database access by means of function calls.

**SQLSetPos** operates on the current rowset and can be used only after a call to **SQLFetchScroll**. The application specifies the number of the row to update, delete, or insert, and the driver retrieves the new data for that row from the rowset buffers. **SQLSetPos** can also be used to designate a specified row as the current row, or to refresh a particular row in the rowset from the data source.

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, **SQLSetPos** is called and then **SQLFetch** or **SQLFetchScroll** is called, and the call to **SQLSetPos** uses the old rowset size while **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.

Because most interaction with relational databases is done through SQL, **SQLSetPos** is not widely supported. However, a driver can easily emulate it by constructing and executing an **UPDATE** or **DELETE** statement.
To determine what operations SQLSetPos supports, an application calls SQLGetInfo with the SQL_DYNAMIC_CURSOR_ATTRIBUTES1, SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1, SQL_KEYSET_CURSOR_ATTRIBUTES1, or SQL_STATIC_CURSOR_ATTRIBUTES1 information option (depending on the type of the cursor).

This section contains the following topics.

- Updating Rows in the Rowset with SQLSetPos
- Deleting Rows in the Rowset with SQLSetPos

© 2016 Microsoft

# Updating Rows in the Rowset with SQLSetPos

The update operation of SQLSetPos makes the data source update one or more selected rows of a table, using data in the application buffers for each bound column (unless the value in the length/indicator buffer is SQL_COLUMN_IGNORE). Columns that are not bound will not be updated.

To update rows with SQLSetPos, the application does the following:

1. Places the new data values in the rowset buffers. For information on how to send long data with SQLSetPos, see Long Data and SQLSetPos and SQLBulkOperations.
2. Sets the value in the length/indicator buffer of each column as necessary. This is the byte length of the data or SQL_NTS for columns bound to string buffers, the byte length of the data for columns bound to binary buffers, and SQL_NULL_DATA for any columns to be set to NULL.
3. Sets the value in the length/indicator buffer of those columns which are not to be updated to SQL_COLUMN_IGNORE. Although the application can skip this step and resend existing data, this is inefficient and risks sending values to the data source that were truncated when they were read.
4. Calls SQLSetPos with Operation set to SQL_UPDATE and RowNumber set to the number of the row to update. If RowNumber is 0, all rows in the rowset are updated.

After SQLSetPos returns, the current row is set to the updated row.

When updating all rows of the rowset (RowNumber is equal to 0), an application can disable the update of certain rows by setting the corresponding elements of the row operation array (pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute) to SQL_ROW_IGNORE. The row operation array corresponds in size and number of elements to the row status array (pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute). To update only those rows in the result set that were successfully fetched and have not been deleted from the rowset, the application uses the row status array from the function that fetched the rowset as the row operation array to SQLSetPos.

For every row that is sent to the data source as an update, the application buffers should have valid row data. 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.

For example, the following code allows a user to scroll through the Customers table and update, delete, or add new rows. It places the new data in the rowset buffers before calling SQLSetPos to update or add new rows. An extra row is allocated at the end of the rowset buffers to hold new rows; this prevents existing data from being overwritten when data for a new row is placed in the buffers.
#define UPDATE_ROW 100
#define DELETE_ROW 101
#define ADD_ROW 102

SQLUINTEGER CustIDArray[11];
SQLCHAR NameArray[11][51], AddressArray[11][51], PhoneArray[11][11];
SQLINTEGER CustIDIndArray[11], NameLenOrIndArray[11], AddressLenOrIndArray[11],
    PhoneLenOrIndArray[11];
SQLUSMALLINT RowStatusArray[10], Action, RowNum;
SQLRETURN rc;
SQLHSTMT hstmt;

// Set the SQL_ATTR_ROW_BIND_TYPE statement attribute to use column-wise
// binding. Declare the rowset size with the SQL_ATTR_ROW_ARRAY_SIZE
// statement attribute. Set the SQL_ATTR_ROW_STATUS_PTR statement
// attribute to point to the row status array.
SQLSetStmtAttr(hstmt, SQL_ATTR_CURSOR_TYPE, SQL_CURSOR_KEYSET_DRIVEN, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, 10, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);

// Bind arrays to the CustID, Name, Address, and Phone columns.
SQLBindCol(hstmt, 1, SQL_C_ULONG, CustIDArray, 0, CustIDIndArray);
SQLBindCol(hstmt, 2, SQL_C_CHAR, NameArray, sizeof(NameArray[0]), NameLenOrIndArray);
SQLBindCol(hstmt, 3, SQL_C_CHAR, AddressArray, sizeof(AddressArray[0]),
    AddressLenOrIndArray);
SQLBindCol(hstmt, 4, SQL_C_CHAR, PhoneArray, sizeof(PhoneArray[0]),
    PhoneLenOrIndArray);

// Execute a statement to retrieve rows from the Customers table.
SQLExecDirect(hstmt, "SELECT CustID, Name, Address, Phone FROM Customers", SQL_NTS);

// Fetch and display the first 10 rows.
rc = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray, AddressArray,
    AddressLenOrIndArray, PhoneArray, PhoneLenOrIndArray, RowStatusArray);

// Call GetAction to get an action and a row number from the user.
while (GetAction(&Action, &RowNum)) {
    switch (Action) {
        case SQL_FETCH_NEXT:
        case SQL_FETCH_PRIOR:
        case SQL_FETCH_FIRST:
        case SQL_FETCH_LAST:
        case SQL_FETCH_ABSOLUTE:
        case SQL_FETCH_RELATIVE:
            // Fetch and display the requested data.
            SQLFetchScroll(hstmt, Action, RowNum);
            DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray,
                AddressArray, AddressLenOrIndArray, PhoneArray, PhoneLenOrIndArray, RowStatusArray);
            break;
        case UPDATE_ROW:
            // Place the new data in the rowset buffers and update the
            // specified row.
            GetNewData(&CustIDArray[RowNum - 1], &CustIDIndArray[RowNum - 1],
                NameArray[RowNum - 1], &NameLenOrIndArray[RowNum - 1],
                AddressArray[RowNum - 1], &AddressLenOrIndArray[RowNum - 1],
                PhoneArray[RowNum - 1], &PhoneLenOrIndArray[RowNum - 1]);
            break;
    }
}
Deleting Rows in the Rowset with SQLSetPos

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. (See Updating Rows in the Rowset with 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.

Updating Data with SQLBulkOperations

Applications can perform bulk update, delete, fetch, or insertion operations on the underlying table at the data source with a call to SQLBulkOperations. Calling SQLBulkOperations is a convenient alternative to constructing and executing an SQL statement. It lets an ODBC driver support positioned updates even when the data source does not support positioned SQL statements. It is part of the paradigm of achieving complete database access by means of function calls.
SQLBulkOperations operates on the current rowset and can be used only after a call to SQLFetch or SQLFetchScroll. The application specifies the rows to update, delete, or refresh by caching their bookmarks. The driver retrieves the new data for rows to be updated, or the new data to be inserted into the underlying table, from the rowset buffers.

The rowset size to be used by SQLBulkOperations is set by a call to SQLSetStmtAttr with an Attribute argument of SQL_ATTR_ROW_ARRAY_SIZE. Unlike SQLSetPos, which uses a new rowset size only after a call to SQLFetch or SQLFetchScroll, SQLBulkOperations uses the new rowset size after the call to SQLSetStmtAttr.

Because most interaction with relational databases is done through SQL, SQLBulkOperations is not widely supported. However, a driver can easily emulate it by constructing and executing an UPDATE, DELETE, or INSERT statement.

To determine what operations SQLBulkOperation supports, an application calls SQLGetInfo with the SQL_DYNAMIC_CURSOR_ATTRIBUTES1, SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1, SQL_KEYSET_CURSOR_ATTRIBUTES1, or SQL_STATIC_CURSOR_ATTRIBUTES1 information option (depending on the type of the cursor).

This section contains the following topics.

- Updating Rows by Bookmark with SQLBulkOperations
- Deleting Rows by Bookmark with SQLBulkOperations
- Inserting Rows with SQLBulkOperations
- Fetching Rows with SQLBulkOperations

© 2016 Microsoft

### Updating Rows by Bookmark with SQLBulkOperations

When updating a row by bookmark, SQLBulkOperations makes the data source update one or more rows of the table. The rows are identified by the bookmark in a bound bookmark column. The row is updated using data in the application buffers for each bound column (except when the value in the length/indicator buffer for a column is SQL_COLUMN_IGNORE). Unbound columns will not be updated.

To update rows by bookmark with SQLBulkOperations, the application:

1. Retrieves and caches the bookmarks of all rows to be updated. If there is more than one bookmark and column-wise binding is used, the bookmarks are stored in an array; if there is more than one bookmark and row-wise binding is used, the bookmarks are stored in an array of row structures.

2. Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of bookmarks and binds the buffer containing the bookmark value, or the array of bookmarks, to column 0.

3. Places the new data values in the rowset buffers. For information on how to send long data with SQLBulkOperations, see [Long Data and SQLSetPos and SQLBulkOperations](#).

4. Sets the value in the length/indicator buffer of each column as necessary. This is the byte length of the data or SQL_NTS for columns bound to string buffers, the byte length of the data for columns bound to binary buffers, and SQL_NULL_DATA for any columns to be set to NULL.

5. Sets the value in the length/indicator buffer of those columns that are not to be updated to SQL_COLUMN_IGNORE. Although the application can skip this step and resend existing data, this is inefficient and risks sending values to the data source that were truncated when they were read.
6. Calls **SQLBulkOperations** with the *Operation* argument set to SQL_UPDATE_BY_BOOKMARK.

For every row that is sent to the data source as an update, the application buffers should have valid row data. If the application buffers were filled by fetching, if a row status array has been maintained, and if the status value for a row is SQL_ROW_DELETED, SQL_ROW_ERROR, or SQL_ROW_NOROW, invalid data could inadvertently be sent to the data source.

© 2016 Microsoft

### Deleting Rows by Bookmark with SQLBulkOperations

When deleting a row by bookmark, **SQLBulkOperations** makes the data source delete one or more selected rows of the table. The rows are identified by the bookmark in a bound bookmark column.

To delete rows by bookmark with **SQLBulkOperations**, the application does the following:

1. Retrieves and caches the bookmarks of all rows to be deleted. If there is more than one bookmark and column-wise binding is used, the bookmarks are stored in an array; if there is more than one bookmark and row-wise binding is used, the bookmarks are stored in an array of row structures.
2. Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of bookmarks and binds the buffer containing the bookmark value, or the array of bookmarks, to column 0.
3. Calls **SQLBulkOperations** with *Operation* set to SQL_DELETE_BY_BOOKMARK.

© 2016 Microsoft

### Inserting Rows with SQLBulkOperations

Inserting data with **SQLBulkOperations** is similar to updating data with **SQLBulkOperations** because it uses data from the bound application buffers.

So that each column in the new row has a value, all bound columns with a length/indicator value of SQL_COLUMN_IGNORE and all unbound columns must either accept NULL values or have a default.

To insert rows with **SQLBulkOperations**, the application does the following:

1. Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows to insert and places the new data values in the bound application buffers. For information on how to send long data with **SQLBulkOperations**, see [Long Data and SQLSetPos and SQLBulkOperations](#).
2. Sets the value in the length/indicator buffer of each column as necessary. This is the byte length of the data or SQL_NTS for columns bound to string buffers, the byte length of the data for columns bound to binary buffers, and SQL_NULL_DATA for any columns to be set to NULL. The application sets the value in the length/indicator buffer of those columns that are to be set to their default (if one exists) or NULL (if one does not) to SQL_COLUMN_IGNORE.
3. Calls **SQLBulkOperations** with the *Operation* argument set to SQL_ADD.

After **SQLBulkOperations** returns, the current row is unchanged. If the bookmark column (column 0) is bound, **SQLBulkOperations** returns the bookmarks of the inserted rows in the rowset buffer bound to that column.
Fetching Rows with SQLBulkOperations

Data can be refetched into a rowset using bookmarks by a call to SQLBulkOperations. The rows to be fetched are identified by the bookmarks in a bound bookmark column. Columns with a value of SQL_COLUMN_IGNORE are not fetched.

To perform bulk fetches with SQLBulkOperations, the application does the following:

1. Retrieves and caches the bookmarks of all rows to be updated. If there is more than one bookmark and column-wise binding is used, the bookmarks are stored in an array; if there is more than one bookmark and row-wise binding is used, the bookmarks are stored in an array of row structures.

2. Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows to fetch and binds the buffer containing the bookmark value, or the array of bookmarks, to column 0.

3. Sets the value in the length/indicator buffer of each column as necessary. This is the byte length of the data or SQL_NTS for columns bound to string buffers, the byte length of the data for columns bound to binary buffers, and SQL_NULL_DATA for any columns to be set to NULL. The application sets the value in the length/indicator buffer of those columns that are to be set to their default (if one exists) or NULL (if one does not) to SQL_COLUMN_IGNORE.

4. Calls SQLBulkOperations with the Operation argument set to SQL_FETCH_BY_BOOKMARK.

There is no need for the application to use the row operation array to prevent the operation to be performed on certain columns. The application selects the rows it wants to fetch by copying only the bookmarks for those rows into the bound bookmark array.

Long Data and SQLSetPos and SQLBulkOperations

As is the case with parameters in SQL statements, long data can be sent when updating rows with SQLBulkOperations or SQLSetPos or when inserting rows with SQLBulkOperations. The data is sent in parts, with multiple calls to SQLPutData. Columns for which data is sent at execution time are known as data-at-execution columns.

Note

An application actually can send any type of data at execution time with SQLPutData, although only character and binary data can be sent in parts. However, if the data is small enough to fit in a single buffer, there is generally no reason to use SQLPutData. It is much easier to bind the buffer and let the driver retrieve the data from the buffer.

Because long data columns typically are not bound, the application must bind the column before calling SQLBulkOperations or SQLSetPos and unbind it after calling SQLBulkOperations or SQLSetPos. The column must be bound because SQLBulkOperations or SQLSetPos operates only on bound columns and must be unbound so that SQLGetData can be used to retrieve data from the column.

To send data at execution time, the application does the following:
1. Places a 32-bit value in the rowset buffer instead of a data value. This value will be returned to the application later, so the application should set it to a meaningful value, such as the number of the column or the handle of a file containing data.

2. Sets the value in the length/indicator buffer to the result of the SQL_LEN_DATA_AT_EXEC(length) macro. This value indicates to the driver that the data for the parameter will be sent with SQLPutData. The length value is used when sending long data to a data source that needs to know how many bytes of long data will be sent so that it can preallocate space. To determine whether a data source requires this value, the application calls SQLGetInfo with the SQL_NEED_LONG_DATA_LEN option. All drivers must support this macro; if the data source does not require the byte length, the driver can ignore it.

3. Calls SQLBulkOperations or SQLSetPos. The driver discovers that a length/indicator buffer contains the result of the SQL_LEN_DATA_AT_EXEC(length) macro and returns SQL_NEED_DATA as the return value of the function.

4. Calls SQLParamData in response to the SQL_NEED_DATA return value. If long data needs to be sent, SQLParamData returns SQL_NEED_DATA. In the buffer pointed to by the ValuePtrPtr argument, the driver returns the unique value that the application placed in the rowset buffer. If there is more than one data-at-execution column, the application uses this value to determine which column to send data for; the driver is not required to request data for data-at-execution columns in any particular order.

5. Calls SQLPutData to send the column data to the driver. If the column data does not fit in a single buffer, as is often the case with long data, the application calls SQLPutData repeatedly to send the data in parts; it is up to the driver and data source to reassemble the data. If the application passes null-terminated string data, the driver or data source must remove the null-termination character as part of the reassembly process.

6. Calls SQLParamData again to indicate that it has sent all of the data for the column. If there are any data-at-execution columns for which data has not been sent, the driver returns SQL_NEED_DATA and the unique value for the next data-at-execution column; the application returns to step 5. If data has been sent for all data-at-execution columns, the data for the row is sent to the data source. SQLParamData then returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO and can return any SQLSTATE that SQLBulkOperations or SQLSetPos can return.

After SQLBulkOperations or SQLSetPos returns SQL_NEED_DATA and before data has been completely sent for the last data-at-execution column, the statement is in a Need Data state. In this state, the application can call only SQLPutData, SQLParamData, SQLCancel, SQLGetDiagField, or SQLGetDiagRec; all other functions return SQLSTATE HY010 (Function sequence error). Calling SQLCancel cancels execution of the statement and returns it to its previous state. For more information, see Appendix B: ODBC State Transition Tables.

© 2016 Microsoft

Descriptors

A descriptor handle refers to a data structure that holds information about either columns or dynamic parameters.

ODBC functions that operate on column and parameter data implicitly set and retrieve descriptor fields. For instance, when SQLBindCol is called to bind column data, it sets descriptor fields that completely describe the binding. When SQLColAttribute is called to describe column data, it returns data stored in descriptor fields.

An application calling ODBC functions need not concern itself with descriptors. No database operation requires that the application gain direct access to descriptors. However, for some applications, gaining direct access to descriptors streamlines many operations. For example, direct access to descriptors provides a way to rebind column data, which can be more efficient than calling SQLBindCol again.

Note

The physical representation of the descriptor is not defined. Applications gain direct access to a descriptor only by manipulating its fields by calling ODBC functions with the descriptor handle.
This section contains the following topics.

- Types of Descriptors
- Descriptor Fields
- Allocating and Freeing Descriptors
- Getting and Setting Descriptor Fields

Types of Descriptors

A descriptor is used to describe one of the following:

- A set of zero or more parameters. A parameter descriptor can be used to describe:
  - The application parameter buffer, which contains either the input dynamic arguments as set by the application or the output dynamic arguments following the execution of a CALL statement of SQL.
  - The implementation parameter buffer. For input dynamic arguments, this contains the same arguments as the application parameter buffer, after any data conversion the application may specify. For output dynamic arguments, this contains the returned arguments, before any data conversion that the application may specify.

For input dynamic arguments, the application must operate on an application parameter descriptor before executing any SQL statement that contains dynamic parameter markers. For both input and output dynamic arguments, the application can specify different data types from those in the implementation parameter descriptor to achieve data conversion.

- A single row of database data. A row descriptor can be used to describe:
  - The implementation row buffer, which contains the row from the database. (These buffers conceptually contain data as written to or read from the database. However, the stored form of database data is not specified. A database could perform additional conversion on the data from its form in the implementation buffer.)
  - The application row buffer, which contains the row of data as presented to the application, following any data conversion that the application may specify.

The application operates on the application row descriptor in any case where column data from the database must appear in application variables. To achieve data conversion of column data, the application can specify different data types from those in the implementation row descriptor.

The descriptor types are summarized in the following table.

<table>
<thead>
<tr>
<th>Buffer type</th>
<th>Rows</th>
<th>Dynamic parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Application row descriptor</td>
<td>Application parameter descriptor (ARD)</td>
</tr>
<tr>
<td>buffer</td>
<td>(ARD)</td>
<td>(APD)</td>
</tr>
<tr>
<td>Implementation</td>
<td>Implementation row descriptor (IRD)</td>
<td>Implementation parameter descriptor (IPD)</td>
</tr>
<tr>
<td>buffer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For either the parameter or the row buffers, if the application specifies different data types in corresponding records of the implementation and application descriptors, the driver performs data conversion when it uses the descriptors. For example, it may convert numeric and datetime values to character-string format. (For valid conversions, see Appendix D: Data Types.)

A descriptor can perform different roles. Different statements can share any descriptor that the application explicitly allocates. A row descriptor in one statement can serve as a parameter descriptor in another statement.

It is always known whether a given descriptor is an application descriptor or an implementation descriptor, even if the descriptor has not yet been used in a database operation. For the descriptors that the implementation implicitly allocates, the implementation records the predefined row relative to the statement handle. Any descriptor that the application allocates by calling SQLAllocHandle is an application descriptor.

© 2016 Microsoft

Descriptor Fields

Descriptors contain header and record fields that completely describe columns or parameters.

A descriptor contains a single copy of the following header fields. Changing a header field affects all columns or parameters.

<table>
<thead>
<tr>
<th>SQL_DESC_ALLOC_TYPE</th>
<th>SQL_DESC_BIND_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_ARRAY_SIZE</td>
<td>SQL_DESC_COUNT</td>
</tr>
<tr>
<td>SQL_DESC_ARRAY_STATUS_PTR</td>
<td>SQL_DESC_ROWS_PROCESSED_PTR</td>
</tr>
<tr>
<td>SQL_DESC_BIND_OFFSET_PTR</td>
<td></td>
</tr>
</tbody>
</table>

A descriptor contains zero or more descriptor records. Each record describes a column or parameter, depending on the type of descriptor. When a new column or parameter is bound, a new record is added to the descriptor. When a column or parameter is unbound, a record is removed from the descriptor. Each record contains a single copy of the following fields:

<table>
<thead>
<tr>
<th>SQL_DESC_AUTO_UNIQUE_VALUE</th>
<th>SQL_DESC_LOCAL_TYPE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_BASE_COLUMN_NAME</td>
<td>SQL_DESC_NAME</td>
</tr>
<tr>
<td>SQL_DESC_BASE_TABLE_NAME</td>
<td>SQL_DESC_NULLABLE</td>
</tr>
<tr>
<td>SQL_DESC_CASE_SENSITIVE</td>
<td>SQL_DESC_OCTET_LENGTH</td>
</tr>
<tr>
<td>SQL_DESC_CATALOG_NAME</td>
<td>SQL_DESC_OCTET_LENGTH_PTR</td>
</tr>
<tr>
<td>SQL_DESC_CONCISE_TYPE</td>
<td>SQL_DESC_PARAMETER_TYPE</td>
</tr>
<tr>
<td>SQL_DESC_DATA_PTR</td>
<td>SQL_DESC_PRECISION</td>
</tr>
<tr>
<td>SQL_DESC_DATETIME_INTERVAL_CODE</td>
<td>SQL_DESC_SCALE</td>
</tr>
<tr>
<td>SQL_DESC_DATETIME_INTERVAL_PRECISION</td>
<td>SQL_DESC_SCHEMA_NAME</td>
</tr>
</tbody>
</table>
Many statement attributes correspond to the header field of a descriptor. Setting these attributes through a call to `SQLSetStmtAttr` and setting the corresponding descriptor header field by calling `SQLSetDescField` have the same effect. The same is true for `SQLGetStmtAttr` and `SQLGetDescField`, both of which retrieve the same information. Calling the statement functions instead of the descriptor functions has the advantage that a descriptor handle does not have to be retrieved.

The following header fields can be set by setting statement attributes:

<table>
<thead>
<tr>
<th>SQL_DESC_ARRAY_SIZE</th>
<th>SQL_DESC_BIND_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_ARRAY_STATUS_PTR</td>
<td>SQL_DESC_ROWS_PROCESSED_PTR</td>
</tr>
<tr>
<td>SQL_DESC_BIND_OFFSET_PTR</td>
<td></td>
</tr>
</tbody>
</table>

This section contains the following topics.

- Record Count
- Bound Descriptor Records
- Deferred Fields
- Consistency Check

© 2016 Microsoft

## Record Count

The SQL_DESC_COUNT header field of a descriptor is the one-based index of the highest-numbered record that contains data. This field is not a count of all columns or parameters that are bound. When a descriptor is allocated, the initial value of SQL_DESC_COUNT is 0.

The driver takes any action necessary to allocate and maintain whatever storage it requires to hold descriptor information. The application does not explicitly specify the size of a descriptor nor allocate new records. When the application provides information for a descriptor record whose number is higher than the value of SQL_DESC_COUNT, the driver automatically increases SQL_DESC_COUNT. When the application unbinds the highest-numbered descriptor record, the driver automatically decreases SQL_DESC_COUNT to contain the number of the highest remaining bound record.
Bound Descriptor Records

When the application sets the SQL_DESC_DATA_PTR field of a descriptor record so that it no longer contains a null value, the record is said to be bound.

If the descriptor is an APD, each bound record constitutes a bound parameter. For input parameters, the application must bind a parameter for each dynamic parameter marker in the SQL statement before executing the statement. For output parameters, the application need not bind the parameter.

If the descriptor is an ARD, which describes a row of database data, each bound record constitutes a bound column.

Deferred Fields

The values of deferred fields are not used when they are set, but the driver saves the addresses of the variables for a deferred effect. For an application parameter descriptor, the driver uses the contents of the variables at the time of the call to SQLExecDirect or SQLExecute. For an application row descriptor, the driver uses the contents of the variables at the time of the fetch.

The following are deferred fields:

- The SQL_DESC_DATA_PTR and SQL_DESC_INDICATOR_PTR fields of a descriptor record.
- The SQL_DESC_OCTET_LENGTH_PTR field of an application descriptor record.
- In the case of a multirow fetch, the SQL_DESC_ARRAY_STATUS_PTR and SQL_DESC_ROWS_PROCESSED_PTR fields of a descriptor header.

When a descriptor is allocated, the deferred fields of each descriptor record initially have a null value. The meaning of the null value is as follows:

- If SQL_DESC_ARRAY_STATUS_PTR has a null value, a multirow fetch fails to return this component of the per-row diagnostic information.
- If SQL_DESC_DATA_PTR has a null value, the record is unbound.
- If the SQL_DESC_OCTET_LENGTH_PTR field of an ARD has a null value, the driver does not return length information for that column.
- If the SQL_DESC_OCTET_LENGTH_PTR field of an APD has a null value and the parameter is a character string, the driver assumes that string is null-terminated. For output dynamic parameters, a null value in this field prevents the driver from returning length information. (If the SQL_DESC_TYPE field does not indicate a character-string parameter, the SQL_DESC_OCTET_LENGTH_PTR field is ignored.)

The application must not deallocate or discard variables used for deferred fields between the time it associates them with the fields and the time the driver reads or writes them.
Consistency Check

A consistency check is performed by the driver automatically whenever an application sets the SQL_DESC_DATA_PTR field of the APD, ARD, or IPD. Whenever this field is set, the driver checks that the value of the SQL_DESC_TYPE field and the values applicable to the SQL_DESC_TYPE field in the same record are valid and consistent.

The SQL_DESC_DATA_PTR field of an IPD is not normally set; however, an application can do so to force a consistency check of IPD fields. The value that the SQL_DESC_DATA_PTR field of the IPD is set to is not actually stored and cannot be retrieved by a call to SQLGetDescField or SQLGetDescRec; the setting is made only to force the consistency check. A consistency check cannot be performed on an IRD.

For more information on the consistency check, see SQLSetDescRec.

Allocating and Freeing Descriptors

Descriptors are either implicitly or explicitly allocated, as described in the following sections.

- Implicitly Allocated Descriptors
- Explicitly Allocated Descriptors
- Initialization of Descriptor Fields
- Automatic Population of the IPD
- Freeing Descriptors

Implicitly Allocated Descriptors

When a statement handle is allocated, the application implicitly allocates one set of four descriptors. The application can obtain the handles of these implicitly allocated descriptors as attributes of the statement handle. When the application frees the statement handle, the driver frees all implicitly allocated descriptors on that handle.
Explicitly Allocated Descriptors

An application can explicitly allocate an application descriptor on a connection at any time it is connected to the database. By specifying that descriptor handle as an attribute of a statement handle using `SQLSetStmtAttr`, the application directs the driver to use that descriptor in place of the corresponding implicitly allocated application descriptors. The application cannot specify alternate implementation descriptors.

An application can associate an explicitly allocated descriptor with more than one statement. Only when an application is actually connected to the database can a descriptor be an explicitly allocated descriptor. The application can free such a descriptor explicitly, or implicitly by freeing its connection.

Initialization of Descriptor Fields

When an application row descriptor is allocated, its fields receive initial values as indicated in `SQLSetDescField`. The initial value of the SQL_DESC_TYPE field is SQL_DEFAULT. This provides for a standard treatment of database data for presentation to the application. The application may specify different treatment of the data by setting fields of the descriptor record.

The initial value of SQL_DESC_ARRAY_SIZE in the descriptor header is 1. The application can modify this field to enable multirow fetch.

The concept of a default value is not valid for the fields of an IRD. An application can gain access to the fields of an IRD only when there is a prepared or executed statement associated with it.

Certain fields of an IPD are defined only after the IPD has been automatically populated by the driver. If not, they are undefined. These fields are SQL_DESC_CASE_SENSITIVE, SQL_DESC_FIXED_PREC_SCALE, SQL_DESC_TYPE_NAME, SQL_DESC_UNSIGNED, and SQL_DESC_LOCAL_TYPE_NAME.

Automatic Population of the IPD

Some drivers are capable of setting the fields of the IPD after a parameterized query has been prepared. The descriptor fields are automatically populated with information about the parameter, including the data type, precision, scale, and other characteristics. This is equivalent to supporting `SQLDescribeParam`. This information can be particularly valuable to an application when it has no other way to discover it, such as when an ad hoc query is performed with parameters that the application does not know about.

An application determines whether the driver supports automatic population by calling `SQLGetConnectAttr` with an `Attribute` of SQL_ATTR_AUTO_IPD. If SQL_TRUE is returned, the driver supports it and the application can enable it by setting the SQL_ATTRIB_ENABLE_AUTO_IPD statement attribute to SQL_TRUE.

When automatic population is supported and enabled, the driver populates the fields of the IPD after an SQL statement containing parameter markers has been prepared by a call to `SQLPrepare`. An application can retrieve this information by calling `SQLGetDescField` or `SQLGetDescRec`, or `SQLDescribeParam`. The application can use the information to bind the most appropriate application buffer for a parameter or to specify a data conversion for it.
Automatic population of the IPD might produce a performance penalty. An application can turn it off by resetting the SQL_ATTR_ENABLE_AUTO_IPD statement attribute to SQL_FALSE (the default value).

© 2016 Microsoft

Freeing Descriptors

Explicitly allocated descriptors can be freed either explicitly, by calling SQLFreeHandle with HandleType of SQL_HANDLE_DESC, or implicitly, when the connection handle is freed. When an explicitly allocated descriptor is freed, all statement handles to which the freed descriptor applied automatically revert to the descriptors implicitly allocated for them.

Implicitly allocated descriptors can be freed only by calling SQLDisconnect, which drops any statements or descriptors open on the connection, or by calling SQLFreeHandle with a HandleType of SQL_HANDLE_STMT to free a statement handle and all the implicitly allocated descriptors associated with the statement. An implicitly allocated descriptor cannot be freed by calling SQLFreeHandle with a HandleType of SQL_HANDLE_DESC.

Even when freed, an implicitly allocated descriptor remains valid, and SQLGetDescField can be called on its fields.

© 2016 Microsoft

Getting and Setting Descriptor Fields

This section describes the methods an application can use to retrieve or set the values in descriptor fields.

This section contains the following topics.

- Obtaining Descriptor Handles
- Retrieving the Values in Descriptor Fields
- Setting Descriptor Fields
- Copying Descriptors
- Using Concise Functions

© 2016 Microsoft

Obtaining Descriptor Handles

An application obtains the handle of any explicitly allocated descriptor as an output argument of the call to SQLAllocHandle. The handle of an implicitly allocated descriptor is obtained by calling SQLGetStmtAttr.
Retrieving the Values in Descriptor Fields

An application can call SQLGetDescField to obtain a single field of a descriptor record. SQLGetDescField gives the application access to all the descriptor fields defined in ODBC, and to driver-defined fields as well.

SQLGetDescRec can be called to retrieve the settings of multiple descriptor fields that affect the data type and storage of column or parameter data.

Setting Descriptor Fields

To modify the fields of a descriptor, an application can call SQLSetDescField. Some fields are read-only and cannot be set. (See the SQLSetDescField function description.)

Descriptor record fields are set with a record number (RecNumber) of 1 or higher, while descriptor header fields are set with a record number of 0. A record number of 0 is also used to set bookmark fields, in accordance with the convention that bookmarks are contained in column 0. This might leave the impression that bookmark fields are contained within the descriptor header, but this is not the case. Bookmark fields are distinct from header fields.

When setting fields individually, the application should follow the sequence defined in SQLSetDescField. Setting some fields causes the driver to set other fields. This ensures that the descriptor is always ready to use once the application has specified a data type. When the application sets the SQL_DESC_TYPE field, the driver checks that other fields that specify the type are valid and consistent.

If a function call that would set a descriptor field fails, the contents of the descriptor field are undefined after the failed function call.

Copying Descriptors

The SQLCopyDesc function is called to copy the fields of one descriptor to another descriptor. Fields can be copied only to an application descriptor or an IPD, but not to an IRD. Fields can be copied from any type of descriptor. Only those fields that are defined for both the source and target descriptors are copied. SQLCopyDesc does not copy the SQL_DESC_ALLOC_TYPE field, because a descriptor’s allocation type cannot be changed. Copied fields overwrite the existing fields.

An ARD on one statement handle can serve as the APD on another statement handle. This allows an application to copy rows between tables without copying data at the application level. To do this, a row descriptor that describes a fetched row of a table is reused as a parameter descriptor for a parameter in an INSERT statement. The SQL_MAX_CONCURRENT_ACTIVITIES information type must be greater than 1 for this operation to succeed.
Using Concise Functions

Some ODBC functions gain implicit access to descriptors. Application writers may find them more convenient than calling `SQLSetDescField` or `SQLGetDescField`. These functions are called concise functions because they perform a number of functions, including setting or getting descriptor fields. Some concise functions let an application set or retrieve several related descriptor fields in a single function call.

Concise functions can be called without first retrieving a descriptor handle for use as an argument. These functions work with the descriptor fields associated with the statement handle that they are called on.

The concise functions `SQLBindCol` and `SQLBindParameter` bind a column or parameter by setting the descriptor fields that correspond to their arguments. Each of these functions performs more tasks than simply setting descriptors. `SQLBindCol` and `SQLBindParameter` provide a complete specification of the binding of a data column or dynamic parameter. An application can, however, change individual details of a binding by calling `SQLSetDescField` or `SQLSetDescRec` and can completely bind a column or parameter by making a series of suitable calls to these functions.

The concise functions `SQLColAttribute`, `SQLDescribeCol`, `SQLDescribeParam`, `SQLNumParams`, and `SQLNumResultCols` retrieve values in descriptor fields. `SQLSetDescRec` and `SQLGetDescRec` are concise functions that, with one call, set or get multiple descriptor fields that affect the data type and storage of column or parameter data. `SQLSetDescRec` is an effective way to change the binding of column or parameter data in one step.

`SQLSetStmtAttr` and `SQLGetStmtAttr` serve as concise functions in some cases. (See [Descriptor Fields](#).)

Diagnostics

Functions in ODBC return diagnostic information in two ways. The return code indicates the overall success or failure of the function, while diagnostic records provide detailed information about the function. At least one diagnostic record — the header record — is returned even if the function succeeds.

Diagnostics information is used at development time to catch programming errors such as invalid handles and syntax errors in hard-coded SQL statements. It is used at run time to catch run-time errors and warnings such as data truncation, access violations, and syntax errors in SQL statements entered by the user.

This section contains the following topics.

- Return Codes
- Diagnostic Records
- Using SQLGetDiagRec and SQLGetDiagField
- Implementing SQLGetDiagRec and SQLGetDiagField
- Diagnostic Handling Examples
Return Codes ODBC

Each function in ODBC returns a code, known as its return code, which indicates the overall success or failure of the function. Program logic is generally based on return codes.

For example, the following code calls SQLFetch to retrieve the rows in a result set. It checks the return code of the function to determine if the end of the result set was reached (SQL_NO_DATA), if any warning information was returned (SQL_SUCCESS_WITH_INFO), or if an error occurred (SQL_ERROR).

```c
SQLRETURN rc;
SQLHSTMT hstmt;

while ((rc=SQLFetch(hstmt)) != SQL_NO_DATA) {
    if (rc == SQL_SUCCESS_WITH_INFO) {
        // Call function to display warning information.
    } else if (rc == SQL_ERROR) {
        // Call function to display error information.
        break;
    }
    // Process row.
}
```

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 following table defines the return codes.

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_SUCCESS</td>
<td>Function completed successfully. The application calls SQLGetDiagField to retrieve additional information from the header record.</td>
</tr>
<tr>
<td>SQL_SUCCESS_WITH_INFO</td>
<td>Function completed successfully, possibly with a nonfatal error (warning). The application calls SQLGetDiagRec or SQLGetDiagField to retrieve additional information.</td>
</tr>
<tr>
<td>SQL_ERROR</td>
<td>Function failed. The application calls SQLGetDiagRec or SQLGetDiagField to retrieve additional information. The contents of any output arguments to the function are undefined.</td>
</tr>
<tr>
<td>SQL_INVALID_HANDLE</td>
<td>Function failed due to an invalid environment, connection, statement, or descriptor handle. This indicates a programming error. No additional information is available from SQLGetDiagRec or SQLGetDiagField. This code is returned only when the handle is a null pointer or is the wrong type, such as when a statement handle is passed for an argument that requires a connection handle.</td>
</tr>
<tr>
<td>SQL_NO_DATA</td>
<td>No more data was available. The application calls SQLGetDiagRec or SQLGetDiagField to retrieve additional information. One or more driver-defined status records in class 02xxx may be returned.</td>
</tr>
</tbody>
</table>

Note
In ODBC 2.x, this return code was named SQL_NO_DATA_FOUND.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_NEED_DATA</td>
<td>More data is needed, such as when parameter data is sent at execution time or additional connection information is required. The application calls SQLGetDiagRec or SQLGetDiagField to retrieve additional information, if any.</td>
</tr>
<tr>
<td>SQL_STILL_EXECUTING</td>
<td>A function that was started asynchronously is still executing. The application calls SQLGetDiagRec or SQLGetDiagField to retrieve additional information, if any.</td>
</tr>
</tbody>
</table>

Diagnostic Records

Associated with each environment, connection, statement, and descriptor handle are *diagnostic records*. These records contain diagnostic information about the last function called that used a particular handle. The records are replaced only when another function is called using that handle. There is no limit to the number of diagnostic records that can be stored at any one time.

There are two types of diagnostic records: a *header record* and zero or more *status records*. The header record is record 0; the status records are records 1 and above. Diagnostic records are composed of a number of separate fields, which are different for the header record and the status records. In addition, ODBC components can define their own diagnostic record fields.

Although diagnostic records can be thought of as structures, there is no requirement for them to actually be structures; how a driver stores the diagnostic information is driver-specific.

Fields in diagnostic records are retrieved with SQLGetDiagField. The SQLSTATE, native error number, and diagnostic message fields of status records can be retrieved in a single call with SQLGetDiagRec.

This section contains the following topics.

- Header Record
- Status Records

Header Record

The 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 the function returns SQL_INVALID_HANDLE. For a complete list of fields in the header record, see the SQLGetDiagField function description.
Status Records

The fields in the status records contain information about specific errors or warnings returned by the Driver Manager, driver, or data source, including the SQLSTATE, native error number, diagnostic message, column number, and row number. Status records can be 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 the SQLGetDiagField function description.

This section contains the following topics.

- Sequence of Status Records
- SQLSTATEs
- Diagnostic Messages

© 2016 Microsoft

Sequence of Status Records

If two or more status records are returned, the Driver Manager and driver rank them according to the following rules. The record with the highest rank is the first record. The source of a record (Driver Manager, driver, gateway, and so on) is not considered when ranking records.

- **Errors** Status records that describe errors have the highest rank. Among error records, records that indicate a transaction failure or possible transaction failure outrank all other records. If two or more records describe the same error condition, SQLSTATEs defined by the Open Group CLI specification (classes 03 through HZ) outrank ODBC-defined and driver-defined SQLSTATEs.

- **Implementation-defined No Data Values** Status records that describe driver-defined No Data values (class 02) have the second highest rank.

- **Warnings** Status records that describe warnings (class 01) have the lowest rank. If two or more records describe the same warning condition, warning SQLSTATEs defined by the Open Group CLI specification outrank ODBC-defined and driver-defined SQLSTATEs.

If there are two or more records with the highest rank, it is undefined which record is the first record. The order of all other records is undefined. In particular, because warnings may appear before errors, applications should check all status records when a function returns a value other than SQL_SUCCESS.

© 2016 Microsoft

SQLSTATEs

SQLSTATEs provide detailed information about the cause of a warning or error. The SQLSTATEs in this manual are based on those found in the ISO/IEF CLI specification, although those SQLSTATEs that start with IM are specific to ODBC.

Unlike return codes, the SQLSTATEs in this manual are guidelines, and drivers are not required to return them. Therefore, while drivers should return the proper SQLSTATE for any error or warning they are capable of detecting,
applications should not count on this always occurring. The reasons for this situation are twofold:

- **Incompleteness** Although this manual lists a large number of errors and warnings and possible causes for those errors and warnings, it is not complete and probably never will be; driver implementations simply vary too much. Any given driver probably will not return all of the SQLSTATEs listed in this manual and might return SQLSTATEs not listed in this manual.

- **Complexity** Some database engines — particularly relational database engines — return literally thousands of errors and warnings. The drivers for such engines are unlikely to map all of these errors and warnings to SQLSTATEs because of the effort involved, the inexactness of the mappings, the large size of the resulting code, and the low value of the resulting code, which often returns programming errors that should never be encountered at run time. Therefore, drivers should map as many errors and warnings as seems reasonable and be sure to map those errors and warnings on which application logic might be based, such as SQLSTATE 01004 (Data truncated).

Because SQLSTATEs are not returned reliably, most applications just display them to the user along with their associated diagnostic message, which is often tailored to the specific error or warning that occurred, and native error code. There is rarely any loss of functionality in doing this, because applications cannot base programming logic on most SQLSTATEs anyway. For example, suppose `SQLExecDirect` returns SQLSTATE 42000 (Syntax error or access violation). If the SQL statement that caused this error is hard-coded or built by the application, this is a programming error and the code needs to be fixed. If the SQL statement is entered by the user, this is a user error and the application has done all that is possible by informing the user of the problem.

When applications do base programming logic on SQLSTATEs, they should be prepared for the SQLSTATE not to be returned or for a different SQLSTATE to be returned. Exactly which SQLSTATEs are returned reliably can be based only on experience with numerous drivers. However, a general guideline is that SQLSTATEs for errors that occur in the driver or Driver Manager, as opposed to the data source, are more likely to be returned reliably. For example, most drivers probably return SQLSTATE HYC00 (Optional feature not implemented), while fewer drivers probably return SQLSTATE 42021 (Column already exists).

The following SQLSTATEs indicate run-time errors or warnings and are good candidates on which to base programming logic. However, there is no guarantee that all drivers return them.

- 01004 (Data truncated)
- 01S02 (Option value changed)
- HY008 (Operation canceled)
- HYC00 (Optional feature not implemented)
- HYT00 (Timeout expired)

SQLSTATE HYC00 (Optional feature not implemented) is particularly significant because it is the only way in which an application can determine whether a driver supports a particular statement or connection attribute.

For a complete list of SQLSTATEs and what functions return them, see Appendix A: ODBC Error Codes. For a detailed explanation of the conditions under which each function might return a particular SQLSTATE, see that function.

© 2016 Microsoft

# Diagnostic Messages

A diagnostic message is returned with each SQLSTATE. The same SQLSTATE is often returned with a number of different messages. For example, SQLSTATE 42000 (Syntax error or access violation) is returned for most errors in SQL syntax. However, each syntax error is likely to be described by a different message.

Sample diagnostic messages are listed in the Error column in the table of SQLSTATEs in Appendix A and in each
function. Although drivers can return these messages, they are more likely to return whatever message is passed to them by the data source.

Applications generally display diagnostic messages to the user, along with the SQLSTATE and native error code. This helps the user and support personnel determine the cause of any problems. The component information embedded in the message is particularly helpful in doing this.

Diagnostic messages come from data sources and components in an ODBC connection, such as drivers, gateways, and the Driver Manager. Typically, data sources do not directly support ODBC. Consequently, if a component in an ODBC connection receives a message from a data source, it must identify the data source as the source of the message. It must also identify itself as the component that received the message.

If the source of an error or warning is a component itself, the diagnostic message must explain this. Therefore, the text of messages has two different formats. For errors and warnings that do not occur in a data source, the diagnostic message must use this format:

```
[ vendor-identifier ][ ODBC-component-identifier ] component-supplied-text
```

For errors and warnings that occur in a data source, the diagnostic message must use this format:

```
[ vendor-identifier ][ ODBC-component-identifier ][ data-source-identifier ] data-source-supplied-text
```

The following table shows the meaning of each element.

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor-identifier</td>
<td>Identifies the vendor of the component in which the error or warning occurred or that received the error or warning directly from the data source.</td>
</tr>
<tr>
<td>ODBC-component-identifier</td>
<td>Identifies the component in which the error or warning occurred or that received the error or warning directly from the data source.</td>
</tr>
<tr>
<td>data-source-identifier</td>
<td>Identifies the data source. For file-based drivers, this is typically a file format, such as Xbase[1] For DBMS-based drivers, this is the DBMS product.</td>
</tr>
<tr>
<td>component-supplied-text</td>
<td>Generated by the ODBC component.</td>
</tr>
<tr>
<td>data-source-supplied-text</td>
<td>Generated by the data source.</td>
</tr>
</tbody>
</table>

[1] In this case, the driver is acting as both the driver and the data source.

Brackets ([ ] ) must be included in the message and do not indicate optional items.

© 2016 Microsoft

**Using SQLGetDiagRec and SQLGetDiagField**

Applications call `SQLGetDiagRec` or `SQLGetDiagField` to retrieve diagnostic information. These functions accept an environment, connection, statement, or descriptor handle and return diagnostics from the function that last used that handle. The diagnostics logged on a particular handle are discarded when a new function is called using that handle. If the function returned multiple diagnostic records, the application calls these functions multiple times; the total number of status records is retrieved by calling `SQLGetDiagField` for the header record (record 0) with the SQL_DIAG_NUMBER option.
Applications retrieve individual diagnostic fields by calling `SQLGetDiagField` and specifying the field to retrieve. Certain diagnostic fields do not have any meaning for certain types of handles. For a list of diagnostic fields and their meanings, see the `SQLGetDiagField` function description.

Applications retrieve the SQLSTATE, native error code, and diagnostic message in a single call by calling `SQLGetDiagRec`; `SQLGetDiagRec` cannot be used to retrieve information from the header record.

For example, the following code prompts the user for an SQL statement and executes it. If any diagnostic information was returned, it calls `SQLGetDiagField` to get the number of status records and `SQLGetDiagRec` to get the SQLSTATE, native error code, and diagnostic message from those records.

```c
SQLCHAR SqlState[6], SQLStmt[100], Msg[SQL_MAX_MESSAGE_LENGTH];
SQLINTEGER NativeError;
SQLSMALLINT i, MsgLen;
SQLRETURN rc1, rc2;
SQLHSTMT hstmt;

// Prompt the user for an SQL statement.
GetSQLStmt(SQLStmt);

// Execute the SQL statement and return any errors or warnings.
rc1 = SQLExecDirect(hstmt, SQLStmt, SQL_NTS);
if (((rc1 == SQL_SUCCESS_WITH_INFO) || (rc1 == SQL_ERROR))
    // Get the status records.
    i = 1;
    while ((rc2 = SQLGetDiagRec(SQL_HANDLE_STMT, hstmt, i, SqlState, &NativeError,
        Msg, sizeof(Msg), &MsgLen)) != SQL_NO_DATA) {
        DisplayError(SqlState, NativeError, Msg, MsgLen);
        i++;
    }
if (((rc1 == SQL_SUCCESS) || (rc1 == SQL_SUCCESS_WITH_INFO))
    // Process statement results, if any.
}
```

Implementing `SQLGetDiagRec` and `SQLGetDiagField`

`SQLGetDiagRec` and `SQLGetDiagField` are implemented by the Driver Manager and each driver. The Driver Manager and each driver maintain diagnostic records for each environment, connection, statement, and descriptor handle, and free those records only when another function is called with that handle or the handle is freed.

Although both the Driver Manager and each driver must determine the first status record according to the rankings in `Sequence of Status Records`, the Driver Manager determines the final sequence of records.

`SQLGetDiagRec` and `SQLGetDiagField` do not post diagnostic records about themselves.

This section contains the following topics.

- Diagnostic Handling Rules
- Role of the Driver Manager
Diagnostic Handling Rules

The following rules govern diagnostic handling in SQLGetDiagRec and SQLGetDiagField.

For all ODBC components:

- Must not replace, alter, or mask errors or warnings received from another ODBC component.
- May add an additional status record when they receive a diagnostic message from another ODBC component. The added record must add real information value to the original message.

For the ODBC component that directly interfaces a data source:

- Must prefix its vendor identifier, its component identifier, and the data source's identifier to the diagnostic message it receives from the data source.
- Must preserve the data source's native error code.
- Must preserve the data source's diagnostic message.

For any ODBC component that generates an error or warning independent of the data source:

- Must supply the correct SQLSTATE for the error or warning.
- Must generate the text of the diagnostic message.
- Must prefix its vendor identifier and its component identifier to the diagnostic message.
- Must return a native error code, if one is available and meaningful.

For the ODBC component that interfaces with the Driver Manager:

- Must initialize the output arguments of SQLGetDiagRec and SQLGetDiagField.
- Must format and return the diagnostic information as output arguments of SQLGetDiagRec and SQLGetDiagField when that function is called.

For one ODBC component other than the Driver Manager:

- Must set the SQLSTATE based on the native error. For file-based drivers and DBMS-based drivers that do not use a gateway, the driver must set the SQLSTATE. For DBMS-based drivers that use a gateway, either the driver or a gateway that supports ODBC may set the SQLSTATE.
The Driver Manager determines the final order in which to return status records that it generates. In particular, it determines which record has the highest rank and is to be returned first. The driver is responsible for ordering status records that it generates. If status records are posted by both the Driver Manager and the driver, the Driver Manager is responsible for ordering them. For more information, see Sequence of Status Records.

The Driver Manager does as much error checking as it can. This saves every driver from checking for the same errors. For example, if a function argument accepts a discrete number of values, such as Operation in SQLSetPos, the Driver Manager checks that the specified value is legal.

The following sections describe the types of conditions checked by the Driver Manager. They are not intended to be exhaustive; for a complete list of the SQLSTATEs the Driver Manager returns, see the "Diagnostics" section of each function; the description of each check made by the Driver Manager starts with the letters "(DM)." Also see the state transition tables in Appendix B: ODBC State Transition Tables; errors shown in parentheses are detected by the Driver Manager.

This section contains the following topics.

- Argument Value Checks
- State Transition Checks
- General Error Checks
- Driver Manager Error and Warning Checks

© 2016 Microsoft

Argument Value Checks

The Driver Manager checks the following types of arguments. Unless otherwise noted, the Driver Manager returns SQL_ERROR for errors in argument values.

- Environment, connection, and statement handles usually cannot be null pointers. The Driver Manager returns SQL_INVALID_HANDLE when it finds a null handle.

- Required pointer arguments, such as OutputHandlePtr in SQLAllocHandle and CursorName in SQLSetCursorName, cannot be null pointers.

- Option flags that do not support driver-specific values must be a legal value. For example, Operation in SQLSetPos must be SQL_POSITION, SQL_REFRESH, SQL_UPDATE, SQL_DELETE, or SQL_ADD.

- Option flags must be supported in the version of ODBC supported by the driver. For example, InfoType in SQLGetInfo cannot be SQL_ASYNC_MODE (introduced in ODBC 3.0) when calling an ODBC 2.0 driver.

- Column and parameter numbers must be greater than 0 or greater than or equal to 0, depending on the function. The driver must check the upper limit of these argument values based on the current result set or SQL statement.

- Length/indicator arguments and data buffer length arguments must contain appropriate values. For example, the argument that specifies the length of a table name in SQLColumns (NameLength3) must be SQL_NT or a value greater than 0; BufferLength in SQLDescribeCol must be greater than or equal to 0. The driver might also need to check these arguments. For example, it might check that NameLength3 is less than or equal to the maximum length of a table name in the data source.
State Transition Checks

The Driver Manager checks that the state of the environment, connection, or statement is appropriate for the function being called. For example, a connection must be in an allocated state when SQLConnect is called; a statement must be in a prepared state when SQLExecute is called. The Driver Manager returns SQL_ERROR for state transition errors.

General Error Checks

The Driver Manager checks one general error. It always returns SQL_ERROR when it encounters the following error: The function must be supported by the driver.

Driver Manager Error and Warning Checks

The Driver Manager completely or partially implements a number of functions and therefore checks for all or some of the errors and warnings in those functions.

- The Driver Manager implements SQLDataSources and SQLDrivers and checks for all errors and warnings in these functions.
- The Driver Manager checks whether a driver implements SQLGetFunctions. If the driver does not implement SQLGetFunctions, the Driver Manager implements and checks for all errors and warnings in it.
- The Driver Manager partially implements SQLAllocHandle, SQLConnect, SQLDriverConnect, SQLBrowseConnect, SQLFreeHandle, SQLGetDiagRec, and SQLGetDiagField and checks for some errors in these functions. It may return the same errors as the driver for some of these functions because both perform similar operations. For example, the Driver Manager or driver may return SQLSTATE IM008 (Dialog failed) if either one is unable to display a login dialog box for SQLDriverConnect.
Role of the Driver

The driver checks for all errors and warnings not checked by the Driver Manager and orders status records that it generates. (An ODBC 2.x driver does not order status records.) This includes errors and warnings in data truncation, data conversion, syntax, and some state transitions. The driver might also check errors and warnings partially checked by the Driver Manager. For example, although the Driver Manager checks whether the value of Operation in SQLSetPos is legal, the driver must check whether it is supported.

The driver also maps native errors — that is, errors returned by the data source — to SQLSTATEs. For example, the driver might map a number of different native errors for illegal SQL syntax to SQLSTATE 42000 (Syntax error or access violation). The driver returns the native error number in the SQL_DIAG_NATIVE field of the status record. Driver documentation should show how errors and warnings are mapped from the data source to arguments in SQLGetDiagRec and SQLGetDiagField.

Diagnostic Handling Examples

The following examples show how various components in an ODBC connection might generate diagnostic messages and how various drivers might return diagnostics to the application with SQLGetDiagRec.

- File-Based Driver Diagnostic Example
- DBMS-Based Driver Diagnostic Example
- Gateways Diagnostic Example
- Driver Manager Diagnostic Example

File-Based Driver Diagnostic Example

A file-based driver acts both as an ODBC driver and as a data source. It can therefore generate errors and warnings both as a component in an ODBC connection and as a data source. Because it also is the component that interfaces with the Driver Manager, it formats and returns arguments for SQLGetDiagRec.

For example, if a Microsoft® driver for dBASE could not allocate sufficient memory, it might return the following values from SQLGetDiagRec:

```
SQLSTATE: "HY001"
Native Error: 42852
Diagnostic Msg: "[Microsoft][ODBC dBASE Driver]Unable to allocate sufficient memory."
```

Because this error was not related to the data source, the driver only added prefixes to the diagnostic message for the vendor ([Microsoft]) and the driver ([ODBC dBASE Driver]).

If the driver could not find the file Employee.dbf, it might return the following values from SQLGetDiagRec:
Because this error was related to the data source, the driver added the file format of the data source ([dBASE]) as a prefix to the diagnostic message. Because the driver was also the component that interfaced with the data source, it added prefixes for the vendor ([Microsoft]) and the driver ([ODBC dBASE Driver]).

© 2016 Microsoft

DBMS-Based Driver Diagnostic Example

A DBMS-based driver sends requests to a DBMS and returns information to the application through the Driver Manager. Because the driver is the component that interfaces with the Driver Manager, it formats and returns arguments for `SQLGetDiagRec`.

For example, if, using SQL/Services, a Microsoft driver for Oracle Rdb encountered an invalid cursor name, it might return the following values from `SQLGetDiagRec`:

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>&quot;42S02&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native Error:</td>
<td>-1305</td>
</tr>
<tr>
<td>Diagnostic Msg:</td>
<td>&quot;[Microsoft][ODBC dBASE Driver][dBASE]No such table or object&quot;</td>
</tr>
</tbody>
</table>

Because the error occurred in the driver, it added prefixes to the diagnostic message for the vendor ([Microsoft]) and the driver ([ODBC Rdb Driver]).

If the DBMS could not find the table EMPLOYEE, the driver might format and return the following values from `SQLGetDiagRec`:

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>&quot;34000&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native Error:</td>
<td>0</td>
</tr>
<tr>
<td>Diagnostic Msg:</td>
<td>&quot;[Microsoft][ODBC Rdb Driver]Invalid cursor name: EMPLOYEE_CURSOR.&quot;</td>
</tr>
</tbody>
</table>

Because the error occurred in the driver, it added prefixes to the diagnostic message for the vendor ([Microsoft]) and the driver ([ODBC Rdb Driver]).

© 2016 Microsoft

Gateways Diagnostic Example

In a gateway architecture, a driver sends requests to a gateway that supports ODBC. The gateway sends the requests to a DBMS. Because it is the component that interfaces with the Driver Manager, the driver formats and returns arguments for `SQLGetDiagRec`.

For example, if Oracle based a gateway to Rdb on Microsoft Open Data Services and if Rdb could not find the table EMPLOYEE, the gateway might generate this diagnostic message:

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>&quot;42S02&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native Error:</td>
<td>-1</td>
</tr>
</tbody>
</table>
| Diagnostic Msg: | "[Microsoft][ODBC Rdb Driver][Rdb] %SQL-F-RELNOTDEF, Table EMPLOYEE "
  "is not defined in schema." |

Because the error occurred in the data source, the driver added a prefix for the data source identifier ([Rdb]) to the diagnostic message. Because the driver was the component that interfaced with the data source, it added prefixes for its vendor ([Microsoft]) and identifier ([ODBC Rdb Driver]) to the diagnostic message.

© 2016 Microsoft
Because the error occurred in the data source, the gateway added a prefix for the data source identifier ([Rdb]) to the diagnostic message. Because the gateway was the component that interfaced with the data source, it added prefixes for its vendor ([DEC]) and identifier ([ODS Gateway]) to the diagnostic message. It also added the SQLSTATE value and the Rdb error code to the beginning of the diagnostic message. This permitted it to preserve the semantics of its own message structure and still supply the ODBC diagnostic information to the driver. The driver parses the error information attached to the error statement by the gateway.

Because the gateway driver is the component that interfaces with the Driver Manager, it would use the preceding diagnostic message to format and return the following values from **SQLGetDiagRec**:

```
SQLSTATE: "42S02"
Native Error: -1
Diagnostic Msg: "[DEC][ODS Gateway][Rdb]%SQL-F-RELNOTDEF, Table EMPLOYEE is not defined in schema."
```

The Driver Manager can also generate diagnostic messages. For example, if an application passed an invalid direction option to **SQLDataSources**, the Driver Manager might format and return the following values from **SQLGetDiagRec**:

```
SQLSTATE: "HY103"
Native Error: 0
Diagnostic Msg: "[Microsoft][ODBC Driver Manager]Direction option out of range"
```

Because the error occurred in the Driver Manager, it added prefixes to the diagnostic message for its vendor ([Microsoft]) and its identifier ([ODBC Driver Manager]).

© 2016 Microsoft

**Driver Manager Diagnostic Example**

Interoperability

**Interoperability** is the ability of a single application to operate with many different DBMSs. The need to write generic, interoperable applications was one of the major factors leading to the development of ODBC. However, interoperability is not a simple path followed from "not interoperable" to "completely interoperable." The path has many branches, and each requires trade-offs among features, speed, code complexity, and development time.

The process of writing an interoperable application follows several steps:

1. Deciding whether the application will use ODBC.
2. Choosing a level of interoperability and deciding which trade-offs are necessary to reach that level.
3. Writing interoperable code and testing it as fully as possible.

It should be noted that interoperability is primarily the domain of the application writer. Drivers are designed to work with a single DBMS and, by definition, are not interoperable. They play a role in interoperability by correctly implementing and exposing ODBC over a single DBMS.

This section contains the following topics.

- Is ODBC the Answer?
- Choosing a Level of Interoperability
- Determining the Target DBMSs and Drivers
- Considering Database Features to Use
- Length of the Product Cycle
- Writing an Interoperable Application
- Testing Interoperable Applications

© 2016 Microsoft

Is ODBC the Answer?

Before delving into the question of interoperability, consider the following question: Should the application use ODBC at all? This might seem a strange question to ask in a guide to ODBC, but it is, in fact, a legitimate one. ODBC was not designed to completely replace native database APIs, nor was it designed to provide database access in all circumstances. It was designed to provide a common interface to databases and was intended to free application programmers from having to learn about and maintain links to multiple databases.

Custom applications are prime candidates for native database APIs. The main reason is that custom applications often work with a single DBMS and have no need to be interoperable. Native database APIs might do a better job than ODBC of exposing the capabilities of a particular DBMS and might expose capabilities not exposed by ODBC. Furthermore, because the developers of custom applications are usually familiar with the native database API for their DBMS, there is little reason to learn ODBC. However, it is interesting to note that for some DBMSs, ODBC is the native database API.

So which applications are candidates for ODBC? The best candidates are applications that work with more than one DBMS. This includes virtually all generic and vertical applications. It also includes a number of custom applications. For example, custom applications that use several different DBMSs are much easier and cleaner to write with ODBC than with multiple native APIs. And custom applications written with ODBC are much easier to migrate as a company moves from one DBMS to another or deploys the same application against different DBMSs.

© 2016 Microsoft

Choosing a Level of Interoperability

Assuming the application will use ODBC, the next step is to determine what level of interoperability is required. The basic level of interoperability is usually a function of the application type: Custom applications tend not to be interoperable, vertical applications tend to be interoperable among a limited number of DBMSs, and generic applications tend to be interoperable among all DBMSs.

This section contains the following topics.
Custom Applications

Custom applications typically perform a specific task for a few DBMSs. For example, an application might retrieve data from a single DBMS and generate a report, or it might transfer data among several DBMSs. What these applications have in common is that these DBMSs are known before the application is written and are unlikely to change over the life of the application.

The custom application therefore requires little or no interoperability. The application developer can choose a single driver for each DBMS and code directly to those drivers. The application can safely contain driver-specific code to exploit the capabilities of those drivers and might even make calls to the native database API to use functionality not supported by ODBC.

The major interoperability concern of most custom applications is whether the target DBMSs will change in the future. If so, this process can be simplified by writing more interoperable code to start with. However, such changing of DBMSs is rare and generally entails a large amount of work. Because of this, developers of custom applications rarely choose to increase interoperability at the expense of functionality; they usually choose to recode that functionality when they change DBMSs.

Vertical Applications

Vertical applications typically perform a well-defined task against a single DBMS. For example, an order entry application tracks the orders in a company. What these types of applications have in common is that the database schema is usually designed by the application developer and, while the application might work with a number of different DBMSs, it works with a single DBMS for a single customer.

Because vertical applications usually require certain functionality, such as scrollable cursors or transactions, they rarely support all DBMSs. Instead, they tend to be highly interoperable among a limited set of DBMSs. Typically, vertical application developers choose to support those DBMSs that represent a large fraction of the market and ignore the rest. They might even choose to support specific drivers for those DBMSs to reduce their testing and product support costs.

Because vertical applications can support a known set of DBMSs, they sometimes contain driver-specific or DBMS-specific code. However, such code is best kept to a minimum because it requires extra time to maintain.

Generic Applications

Generic applications sometimes perform a hard-coded task, such as a spreadsheet retrieving data from a database. They might also perform a variety of user-defined tasks, such as a generic query application allowing the user to enter and execute an SQL statement. What generic applications have in common is that they must work with a variety of different DBMSs and that the developer does not know beforehand what these DBMSs will be.
Therefore, generic applications need to be highly interoperable. The developer must make many choices, trading off interoperability for features, and must write code that expects drivers to support a wide range of functionality. While generic applications might be tuned to work with popular DBMSs, they rarely contain driver-specific or DBMS-specific code.

© 2016 Microsoft

Determining the Target DBMSs and Drivers

The next question to consider is, what are the target DBMSs for the application, and what drivers are available that support those DBMSs? Because generic applications tend to be highly interoperable, the question of target DBMSs is most applicable to custom and vertical applications. However, the question of target drivers applies to all applications, because drivers vary widely in speed, quality, feature support, and availability. Also, if drivers are to be redistributed with the application, the cost and availability of licensing plans need to be considered.

For many custom applications, the target DBMSs are obvious: They are existing DBMSs that the application is designed to access. DBMSs to which future migration is planned should also be considered. However, the major question for these applications is which driver or drivers to use with them. For other custom applications — those which are not designed to access an existing DBMS — the target DBMSs can be chosen based on feature support, concurrent user support, driver availability, and affordability.

For vertical applications, the target DBMSs are usually chosen based on feature support, driver availability, and market. For example, a vertical application designed for small businesses must target DBMSs that are affordable to those businesses; a vertical application designed as an add-on to existing DBMSs must target widely used DBMSs.

When choosing target DBMSs, the differences between desktop and server databases should be considered. Desktop databases such as dBASE, Paradox, and Btrieve are less powerful than server databases. Because they are generally accessed through the less powerful SQL engines found in most file-based drivers, they often lack full transaction support, support fewer concurrent users, and have limited SQL. However, they are inexpensive and have a large installed base.

Server databases such as Oracle, DB2, and SQL Server provide full transaction support, support many concurrent users, and have rich SQL. They are much more expensive and have a smaller installed base. On the other hand, software prices tend to be higher, somewhat offsetting a smaller potential market.

Thus, target DBMSs sometimes can be chosen based on the features required by the application and the application's target market. For example, an order entry system for large corporations might not target desktop databases because these lack adequate transaction support. A similar system designed for small businesses might exclude most server databases on the basis of cost. And developers of generic applications might target both but avoid using the advanced features found in server databases.

© 2016 Microsoft

Considering Database Features to Use

After the basic level of interoperability is known, the database features used by the application must be considered. For example, what SQL statements will the application execute? Will the application use scrollable cursors? Transactions? Procedures? Long data? For ideas about what features might not be supported by all DBMSs, see the SQLGetInfo, SQLSetConnectAttr, and SQLSetStmtAttr function descriptions, and Appendix C: SQL Grammar. The features required by an application might eliminate some DBMSs from the list of target DBMSs. They might also show that the application can easily target many DBMSs.

For example, if the required features are simple, they can usually be implemented with a high degree of interoperability. An application that executes a simple SELECT statement and retrieves results with a forward-only
cursor is likely to be highly interoperable by virtue of its simplicity: Almost all drivers and DBMSs support the functionality it needs.

However, if the required features are more complex, such as scrollable cursors, positioned update and delete statements, and procedures, trade-offs must often be made. There are several possibilities:

- **Lower interoperability, more features.** The application includes the features but works only with DBMSs that support them.
- **Higher interoperability, fewer features.** The application drops the features but works with more DBMSs.
- **Higher interoperability, optional features.** The application includes the features but makes them available only with those DBMSs that support them.
- **Higher interoperability, more features.** The application uses the features with DBMSs that support them and emulates them for DBMSs that do not.

The first two cases are relatively simple to implement, because the features are used either with all supported DBMSs or with none. The latter two cases, on the other hand, are more complex. It is necessary in both cases to check whether the DBMS supports the features and in the last case to write a potentially large amount of code to emulate these features. Therefore, these schemes are likely to require more development time and may be slower at run time.

Consider a generic query application that can connect to a single data source. The application accepts a query from the user and displays the results in a window. Now suppose this application has one feature that allows users to display the results of multiple queries simultaneously. That is, they can execute a query and look at some of the results, execute a different query and look at some of its results, and then return to the first query. This presents an interoperability problem because some drivers support only a single active statement.

The application has a number of choices, based on what the driver returns for the `SQL_MAX_CONCURRENT_ACTIVITIES` option in `SQLGetInfo`:

- **Always support multiple queries.** After connecting to a driver, the application checks the number of active statements. If the driver supports only one active statement, the application closes the connection and informs the user that the driver does not support required functionality. The application is easy to implement and has full functionality but has lower interoperability.

- **Never support multiple queries.** The application drops the feature altogether. It is easy to implement and has high interoperability but has less functionality.

- **Support multiple queries only if the driver does.** After connecting to a driver, the application checks the number of active statements. The application allows the user to start a new statement when one is already active only if the driver supports multiple active statements. The application has higher functionality and interoperability but is harder to implement.

- **Always support multiple queries and emulate them when necessary.** After connecting to a driver, the application checks the number of active statements. The application always allows the user to start a new statement when one is already active. If the driver supports only one active statement, the application opens an additional connection to that driver and executes the new statement on that connection. The application has full functionality and high interoperability but is harder to implement.

© 2016 Microsoft

**Length of the Product Cycle**

The final question about interoperability is time. Developing an interoperable application usually takes longer than
developing a noninteroperable one. The reason is that the application must check DBMS capabilities, perform the same tasks differently for different DBMSs, work around functionality supported by some DBMSs but not others, and so on.

In addition to development time, product lifetime must be considered. If the application is designed to be used once, such as an application that transfers data when migrating from one DBMS to another, there is no point in making it interoperable. The application will be used once and discarded.

If the application will exist for a long time, it might be easier to maintain as an interoperable application. This is true even for custom applications that have a single DBMS as a target. The reason is that interoperable code uses a limited subset of database features. The driver is required to keep those features available, even in the face of changes to the underlying DBMS. Thus, interoperable code can shift the burden of coping with changes to the DBMS from the application developer to the driver developer.

Writing an Interoperable Application

Whenever an application uses the same code against more than one driver, that code must be interoperable among those drivers. In most cases, this is an easy task. For example, the code to fetch rows with a forward-only cursor is the same for all drivers. In some cases, this can be more difficult. For example, the code to construct identifiers for use in SQL statements needs to consider identifier case, quoting, and one-part, two-part, and three-part naming conventions.

In general, interoperable code must cope with problems of feature support and feature variability. Feature support refers to whether or not a particular feature is supported. For example, not all DBMSs support transactions, and interoperable code must work correctly regardless of transaction support. Feature variability refers to variation in the manner in which a particular feature is supported. For example, catalog names are placed at the start of identifiers in some DBMSs and at the end of identifiers in others.

Applications can deal with feature support and feature variability at design time or at run time. To deal with feature support and variability at design time, a developer looks at the target DBMSs and drivers and makes sure that the same code will be interoperable among them. This is generally the way in which applications with low or limited interoperability deal with these problems.

For example, if the developer guarantees that a vertical application will work only with four particular DBMSs and if each of those DBMSs supports transactions, the application does not need code to check for transaction support at run time. It can always assume transactions are available because of the design-time decision to use only four DBMSs, each of which supports transactions.

To deal with feature support and variability at run time, the application must test for different capabilities at run time and act accordingly. This is generally the way in which highly interoperable applications deal with these problems. For feature support problems, this means writing code that makes the feature optional or writing code that emulates the feature when it is not available. For feature variability problems, this means writing code that supports all possible variations.

This section contains the following topics.

- Checking Feature Support and Variability
- Features to Watch For
To check feature support and variability, applications generally call `SQLGetInfo`, `SQLGetFunctions`, and `SQLGetTypeInfo`. A good starting place is the driver’s API and SQL grammar conformance levels. These describe broad levels of feature support. The application can then call `SQLGetInfo` with other options to determine the support or variability of features it needs, `SQLGetFunctions` to determine whether functions it needs beyond the returned conformance level are supported, and `SQLGetTypeInfo` to determine what SQL data types are supported.

An application can determine whether a statement or connection attribute is supported by calling `SQLSetStmtAttr` or `SQLSetConnectAttr` with that attribute. If the function returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the attribute is supported; if it returns SQL_ERROR and SQLSTATE HYC00 (Optional feature not implemented), the attribute is not supported.

Applications can also determine a limited amount of information before connecting to the driver by calling `SQLDrivers`.

© 2016 Microsoft

Features to Watch For

This section describes a number of features that application developers often take for granted. In fact, these features vary widely in support and manner of support among DBMSs; failure to code for them is likely to cause problems in interoperable applications.

This section does not list all features that application developers need to consider. For that information, see the `SQLGetInfo`, `SQLSetStmtAttr`, and `SQLSetConnectAttr` function descriptions, Appendix C: SQL Grammar, and the sections of this manual that discuss each feature.

This section contains the following topics.

- Version Number
- Multiple Active Statements and Connections
- Transaction Support in DBMSs
- Commit and Rollback Behavior
- NOT NULL in CREATE TABLE Statements
- Supported Data Types
- ODBC SQL Grammar
- Batch Processing

© 2016 Microsoft

Version Number

There are several versions of ODBC, each with different features. An application determines which ODBC version the Driver Manager and a particular driver support by calling `SQLGetInfo` with the SQL_ODBC_VER and SQL_DRIVER_ODBC_VER options.
Multiple Active Statements and Connections

Some drivers and DBMSs limit the number of statements and connections that can be active at one time. These numbers can be as small as one. For more information, see the SQL_MAX_CONCURRENT_ACTIVITIES and SQL_MAX_DRIVER_CONNECTIONS options in the SQLGetInfo function description, and Statement Handles and Connection Handles.

Transaction Support in DBMSs

Some databases, especially desktop databases such as dBASE, Paradox, and Btrieve, do not support transactions. Even among databases that support transactions, there is variation in what kinds of SQL statements can be in a transaction. For more information, see the SQL_TXN_CAPABLE option in the SQLGetInfo function description.

Commit and Rollback Behavior

A common behavior among server DBMSs is to close cursors and discard prepared statements when a statement is committed or rolled back. Desktop databases are more likely to keep cursors open and keep prepared statements. For more information, see the SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR options in the SQLGetInfo function description and Effect of Transactions on Cursors and Prepared Statements.
NOT NULL in CREATE TABLE Statements

Some databases, and especially desktop databases, do not support the **NOT NULL** column constraint in **CREATE TABLE** statements. For more information, see the SQL_NON_NULLABLE_COLUMNS option in the `SQLGetInfo` function description.

Supported Data Types

The data types supported by DBMSs vary considerably. An application can determine the names and characteristics of supported data types by calling `SQLGetTypeInfo`. Because of wide variation in data type names, the application must use the data type names returned by `SQLGetTypeInfo` in **CREATE TABLE** statements. For more information, see Data Types in ODBC.

ODBC SQL Grammar

Interoperable applications should always use the ODBC SQL grammar in SQL statements. However, considerable variation is possible even within this grammar. For more information, see Interoperability of SQL Statements.
Batch Processing

Support for batches of SQL statements is not widespread, so interoperable applications should use them conditionally or not at all. For more information, see Executing Batches.

Testing Interoperable Applications

Testing interoperable applications is at best a time-consuming business and at worst impossible because new drivers continually appear on the market. However, a reasonable degree of testing is possible. Applications with limited or low interoperability need only be tested against those drivers they are guaranteed to support. However, they must be fully tested against these drivers.

Highly interoperable applications cannot be tested practically against all drivers. The best that most application developers can do is to test them fully against a small number of drivers and cursorily against several more. Tested drivers should include the most popular drivers for the most popular DBMSs in the application’s market; if the market covers all DBMSs, drivers for both desktop and server DBMSs should be tested.

One of the problems in testing ODBC applications is the number of components involved: the application itself, the Driver Manager, the driver, the DBMS, and possibly network software or gateways. Applications can make it easier to track errors by posting the error messages returned by ODBC functions through SQLGetDiagField and SQLGetDiagRec. These messages identify the manufacturer and component in which errors occur. For more information, see Diagnostics.

Programming Considerations

This section briefly discusses a number of topics related to writing ODBC applications and drivers.

This section contains the following topics.

- Multithreading
- Alignment
- Unicode
- Translation DLLs
- Diagnostic Tools
Multithreading

On multithread operating systems, drivers must be thread-safe. That is, it must be possible for applications to use the same handle on more than one thread. How this is achieved is driver-specific, and it is likely that drivers will serialize any attempts to concurrently use the same handle on two different threads.

Applications commonly use multiple threads instead of asynchronous processing. The application creates a separate thread, calls an ODBC function on it, and then continues processing on the main thread. Rather than having to continually poll the asynchronous function, as is the case when the SQL_ATTR_ASYNC_ENABLE statement attribute is used, the application can simply let the newly created thread finish.

Functions that accept a statement handle and are running on one thread can be canceled by calling SQLCancel with the same statement handle from another thread. Although drivers should not serialize the use of SQLCancel in this manner, there is no guarantee that calling SQLCancel will actually cancel the function running on the other thread.

Alignment

The alignment issues in an ODBC application are generally no different than they are in any other application. That is, most ODBC applications have few or no problems with alignment. The penalties for not aligning addresses vary with the hardware and operating system and might be as minor as a slight performance penalty or as major as a fatal run-time error. Therefore, ODBC applications, and portable ODBC applications in particular, should be careful to align data properly.

One example of when ODBC applications encounter alignment issues is when they allocate a large block of memory and bind different parts of that memory to the columns in a result set. This is most likely to occur when a generic application must determine the shape of a result set at run time and allocate and bind memory accordingly.

For example, suppose an application executes a SELECT statement entered by the user and fetches the results from this statement. Because the shape of this result set is not known when the program is written, the application must determine the type of each column after the result set is created and bind memory accordingly. The easiest way to do this is to allocate a large block of memory and bind different addresses in that block to each column. To access the data in a column, the application casts the memory bound to that column.

The following diagram shows a sample result set and how a block of memory might be bound to it using the default C data type for each SQL data type. Each "X" represents a single byte of memory. (This example shows only the data buffers that are bound to the columns. This is done for simplicity. In actual code, the length/indicator buffers must also be aligned.)
Assuming the bound addresses are stored in the Address array, the application uses the following expressions to access the memory bound to each column:

\[
\begin{align*}
\text{(SQLCHAR *) Address[0]} \\
\text{(SQLSMALLINT *) Address[1]} \\
\text{(SQLINTEGER *) Address[2]}
\end{align*}
\]

Notice that the addresses bound to the second and third columns start on odd-numbered bytes and that the address bound to the third column is not divisible by four, which is the size of an SDWORD. On some machines, this will not be a problem; on others, it will cause a slight performance penalty; on still others, it will cause a fatal run-time error. A better solution would be to align each bound address on its natural alignment boundary. Assuming this is 1 for a UCHAR, 2 for an SWORD, and 4 for an SDWORD, this would give the result shown in the following illustration, where an “X” represents a byte of memory that is used and an “O” represents a byte of memory that is unused.

While this solution does not use all of the application’s memory, it does not encounter any alignment problems. Unfortunately, it takes a fair amount of code to implement this solution, as each column must be aligned individually according to its type. A simpler solution is to align all columns on the size of the largest alignment boundary, which is 4 in the example shown in the following illustration.

Although this solution leaves larger holes, the code to implement it is relatively simple and fast. In most cases, this offsets the penalty paid in unused memory. For an example that uses this method, see Using SQLBindCol.
Unicode defines encoding for characters in many languages.

For more information about the Unicode standard, see The Unicode Consortium.

Unicode defines a universal character set. A Windows ANSI code page defines a character set, typically containing characters for one language. It may be more difficult to write an application that is required to use different code pages.

Unicode does not require a code page. Every code point is mapped to a single character in some language.

Currently, the only Unicode encoding that ODBC supports is UCS-2, which uses a 16-bit integer (fixed length) to represent a character. Unicode allows applications to work in different languages.

The ODBC 3.5 (or higher) Driver Manager is Unicode-enabled. This affects two major areas: function calls and string data types. The Driver Manager maps function string arguments and string data as required by the application and driver, both of which can be either Unicode-enabled or ANSI-enabled. These two areas are discussed in detail in the sections, Unicode Function Arguments and Unicode Data.

The ODBC 3.5 (or higher) Driver Manager supports the use of a Unicode driver with both a Unicode application and an ANSI application. It also supports the use of an ANSI driver with an ANSI application. The Driver Manager provides limited Unicode-to-ANSI mapping for a Unicode application working with an ANSI driver.

This section contains the following topics.

- Unicode Function Arguments
- Unicode Data

**Unicode Function Arguments**

The ODBC 3.5 (or higher) Driver Manager supports both ANSI and Unicode versions of all functions that accept pointers to character strings or SQLPOINTER in their arguments. The Unicode functions are implemented as functions (with a suffix of W), not as macros. The ANSI functions (which can be called with or without a suffix of A) are identical to the current ODBC API functions.

**Remarks**

Unicode functions that always return or take strings or length arguments are passed as count-of-characters. For functions that return length information for server data, the display size and precision are described in number of characters. When a length (transfer size of the data) could refer to string or nonstring data, the length is described in octet lengths. For example, SQLGetInfoW will still take the length as count-of-bytes, but SQLExecDirectW will use count-of-characters.

Count-of-characters refers to the number of bytes (octets) for ANSI functions and the number of WCHAR (16-bit words) for UNICODE functions. In particular, a double-byte character sequence (DBCS) or a multibyte character sequence (MBCS) can be composed of multiple bytes. A UTF-16 Unicode character sequence can be composed of multiple WCHARs.

The following is a list of the ODBC API functions that support both Unicode (W) and ANSI (A) versions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLBrowseConnect</td>
<td>SQLGetDiagField</td>
</tr>
<tr>
<td>SQLColAttribute</td>
<td>SQLGetDiagRec</td>
</tr>
</tbody>
</table>
The following is a list of the ODBC Installer and ODBC Translator functions that support both Unicode (W) and ANSI (A) versions:

<table>
<thead>
<tr>
<th>SQLConfigDataSource</th>
<th>SQLInstallDriverManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCreateDataSource</td>
<td>SQLInstallerError</td>
</tr>
<tr>
<td>SQLDataSourcesToDriver</td>
<td>SQLInstallODBC</td>
</tr>
<tr>
<td>SQLDriverToDataSource</td>
<td>SQLReadFileDSN</td>
</tr>
<tr>
<td>SQLGetAvailableDrivers</td>
<td>SQLRemoveDSNFromINI</td>
</tr>
<tr>
<td>SQLGetInstalledDrivers</td>
<td>SQLValidDSN</td>
</tr>
<tr>
<td>SQLGetTranslator</td>
<td>SQLWriteDSNToINI</td>
</tr>
<tr>
<td>SQLInstallDriver</td>
<td></td>
</tr>
</tbody>
</table>

Note
This section contains the following topics.

- Unicode Applications
- Unicode Drivers
- Function Mapping in the Driver Manager

© 2016 Microsoft

Unicode Applications

You can recompile an application as a Unicode application in one of two ways:

- Include the Unicode `#define` contained in the Sqlucode.h header file in the application.
- Compile the application with the compiler's Unicode option. (This option will be different for different compilers.)

To convert an ANSI application to a Unicode application, write the application to store and pass Unicode data. In addition, calls to functions that support SQLPOINTER arguments must be converted to use count of bytes.

After an application is compiled as a Unicode application, if the application calls an ODBC API function (without a suffix), the Driver Manager recognizes the application as a Unicode application and converts the function call to a Unicode function (with the W suffix) if the underlying driver supports Unicode. When an ANSI application makes a function call without a suffix, the Driver Manager converts it to ANSI if the underlying driver supports ANSI. If both the application and the driver support the same character encoding, the driver manager passes the calls through to the driver (with certain exceptions for ANSI applications).

An application can call both Unicode functions (with the W suffix) and ANSI functions (with or without the A suffix). Unicode and ANSI function calls can be mixed. If the cursor library is to be used, however, Unicode and ANSI function calls cannot be mixed. The cursor library is either Unicode or ANSI, not a mixture.

An application can be written such that it can be compiled as either a Unicode application or an ANSI application. In this case, character data types can be declared as SQL_C_TCHAR. This is a macro that inserts SQL_C_WCHAR if the application is compiled as a Unicode application or inserts SQL_C_CHAR if it is compiled as an ANSI application. The application programmer must be careful of functions that take SQLPOINTER as their argument, because the size of the length argument will change (for string data types) depending on whether the application is ANSI or Unicode.

A function can be called in one of three ways: as a Unicode-only function call (with the W suffix), as an ANSI-only function call (with the A suffix), or as the ODBC function call with no suffix. The arguments to the three forms of a function are identical. Only those functions with SQLCHAR * arguments or SQLPOINTER arguments that point to strings require Unicode and ANSI forms. For functions that have arguments that can be declared as a character type, such as `SQLBindCol` or `SQLGetData` (which do not have Unicode and ANSI forms), the argument can be declared as the Unicode type, the ANSI type, or in the case of a C type argument, the SQL_C_TCHAR macro. For more information, see Unicode Data.

An application can be written as a Unicode application even if no Unicode drivers are available for it to work with. The Driver Manager will map Unicode functions and data types to ANSI. There are some restrictions to the Unicode to ANSI mappings that can be performed. The existence of a Unicode driver for the Unicode application to work with will result in better performance and will remove the restrictions inherent in the Unicode to ANSI mappings.
Unicode Drivers

Whether a driver should be a Unicode driver or an ANSI driver depends entirely on the nature of the data source. If the data source supports Unicode data, the driver should be a Unicode driver. If the data source only supports ANSI data, the driver should remain an ANSI driver.

A Unicode driver must export `SQLConnectW` to be recognized as a Unicode driver by the Driver Manager.

A Unicode driver must accept Unicode functions (with a suffix of `W`) and store Unicode data. It can also accept ANSI functions, but is not required to. (The Driver Manager does not pass an ANSI function call with the `A` suffix to the driver, but converts it to an ANSI function call without the suffix and then passes it to the driver.)

A Unicode driver must be able to return result sets in either Unicode or ANSI, depending on the application’s binding. If an application binds to `SQL_C_CHAR`, the Unicode driver must convert SQL_WCHAR data to SQL_CHAR. The driver manager will map SQL_C_WCHAR to SQL_C_CHAR for ANSI drivers but does no mapping for Unicode drivers.

**Note**

When determining the driver type, the Driver Manager will call `SQLSetConnectAttr` and set the `SQL_ATTR_ANSI_APP` attribute at connection time. If the application is using ANSI APIs, `SQL_ATTR_ANSI_APP` will be set to `SQL_AA_TRUE`, and if it is using Unicode, it will be set to a value of `SQL_AA_FALSE`. This attribute is used so that the driver can exhibit different behavior based on the application type. The attribute cannot be set by the application directly, and it is not supported by `SQLGetConnectAttr`. If a driver exhibits the same behavior for both ANSI and Unicode applications, it should return SQL_ERROR for this attribute. If the driver returns SQL_SUCCESS, the Driver Manager will separate ANSI and Unicode connections when Connection Pooling is used.

Function Mapping in the Driver Manager

The driver manager supports two entry points for functions that take string arguments. The undecorated function (`SQLDriverConnect`) is the ANSI form of the function. The Unicode form is decorated with a `W` (`SQLDriverConnectW`).

The ODBC header file also supports functions decorated with an `A`, (`SQLDriverConnectA`) for the convenience of mixed ANSI/Unicode applications. Calls made to the `A` functions are actually calls into the undecorated entry point (`SQLDriverConnect`).

If the application is compiled with the `_UNICODE` #define, the ODBC header file will map undecorated function calls (`SQLDriverConnect`) to the Unicode version (`SQLDriverConnectW`).

The Driver Manager recognizes a driver as a Unicode driver if `SQLConnectW` is supported by the driver.

If the driver is a Unicode driver, the Driver Manager makes function calls as follows:

- Passes a function without string arguments or parameters directly through to the driver.
- Passes Unicode functions (with the `W` suffix) directly through to the driver.
- Converts an ANSI function (with the `A` suffix) to a Unicode function (with the `W` suffix) by converting the string arguments into Unicode characters and passes the Unicode function to the driver.

If the driver is an ANSI driver, the Driver Manager makes function calls as follows:
Passes functions without string arguments or parameters directly through to the driver.

- Converts Unicode functions (with the W suffix) to an ANSI function call and passes it to the driver.
- Passes an ANSI function directly to the driver.

The Driver Manager is Unicode-enabled internally. As a result, the optimum performance is obtained by a Unicode application working with a Unicode driver, because the Driver Manager simply passes Unicode functions through to the driver. When an ANSI application is working with an ANSI driver, the Driver Manager must convert strings from ANSI to Unicode when processing some functions, such as `SQLDriverConnect`. After processing the function, the Driver Manager must then convert the Unicode string back to ANSI before sending the function to the ANSI driver.

An application should not modify or read its bound parameter buffers when the driver returns `SQL_STILL_EXECUTING` or `SQL_NEED_DATA`. The Driver Manager leaves the buffers bound to ANSI until the driver returns `SQL_SUCCESS`, `SQL_SUCCESS_WITH_INFO`, or `SQL_ERROR`. A multithreaded application should not gain access to any bound parameter values that another thread is executing an SQL statement on. The Driver Manager converts the data from Unicode to ANSI "in place," and the other thread might see ANSI data in these buffers while the driver is still processing the SQL statement. Applications that bind Unicode data to an ANSI driver must not bind two different columns to the same address.

Unicode Data

SQL Unicode data types are provided to describe data that resides in Unicode natively on the DBMS. A C Unicode data type is provided to allow an application to bind data to a Unicode buffer. The Driver Manager can convert data from a Unicode C type (SQL_C_WCHAR) to make it function with an ANSI driver.

An ODBC 3.0 or 2.x application will always bind to the ANSI data types. For optimum performance, an ODBC 3.5 (or higher) application should bind to the ANSI data C type if the SQL column type is ANSI, and should bind to the Unicode C data type if the SQL column type is Unicode.

The SQL Unicode type indicators are SQL_WCHAR, SQL_WVARCHAR, and SQL_WLONGVARCHAR. SQL_WCHAR data has a fixed string length, while SQL_WVARCHAR has a variable length with a declared maximum and SQL_WLONGVARCHAR has a variable length with a maximum that depends on the data source.

The C Unicode type indicator is SQL_C_WCHAR. This is the default for each of the SQL Unicode type indicators. All of the SQL types can be converted to SQL_C_WCHAR, and SQL_C_WCHAR can be converted to all of the SQL types. An application can retrieve data in one of three ways:

- Retrieve the data as SQL_C_CHAR.
- Retrieve the data as SQL_C_WCHAR.
- Declare the data as SQL_C_TCHAR. This is a macro that inserts SQL_C_WCHAR if the application is compiled as a Unicode application or inserts SQL_C_CHAR if it is compiled as an ANSI application.

SQL_C_TCHAR is declared in a function as follows:

```c
SQLBindParameter(StatementHandle, 1, SQL_PARAM_INPUT, SQL_C_TCHAR, SQL_WCHAR, NameLen, 0, Name, 0, &a
```

When the application is compiled as a Unicode application, the `ValueType` argument would be changed from SQL_C_TCHAR to SQL_C_WCHAR. When the application is compiled as an ANSI application, the `ValueType` argument would be changed to SQL_C_CHAR.

Unicode drivers must still support ANSI data types, including SQL_CHAR. If an application working with a Unicode
driver binds to SQL_CHAR, the Driver Manager will not map the SQL_CHAR data to SQL_WCHAR. The Unicode driver must accept the SQL_CHAR data.

The Driver Manager stores driver and DSN names in Unicode and maps them to ANSI as needed. If a Unicode character cannot be mapped to an ANSI character (as can occur if characters from a code page that is not the native code page of the computer are used in driver and DSN names), the characters that could not be converted are represented by a default character supplied by the system.

Translation DLLs

The application and data source often store data in different character sets. ODBC provides a generic mechanism that allows the driver to translate data from one character set to another. It consists of a DLL that implements the translation functions SQLDriverToDataSource and SQLDataSourceToDriver, which are called by the driver to translate all data flowing between the data source and driver. This DLL can be written by the application developer, the driver developer, or a third party.

The translation DLL for a particular data source can be specified in the system information for that data source; for more information, see Data Source Specification Subkeys. It can also be set at run time with the SQL_ATTR_TRANSLATE_DLL and SQL_ATTR_TRANSLATE_OPTION connection attributes.

The translation option is a value that can be interpreted only by a particular translation DLL. For example, if the translation DLL translates between different code pages, the option might give the numbers of the code pages used by the application and the data source. There is no requirement for a translation DLL to use a translation option.

After a translation DLL has been specified, the driver loads it and calls it to translate all data flowing between the application and data source. This includes all SQL statements and character parameters being sent to the data source, and all character results, character metadata such as column names, and error messages retrieved from the data source. Connection data is not translated, because the translation DLL is not loaded until after the application has connected to the data source.

Diagnostic Tools

Two facilities can assist with diagnosing problems in ODBC applications. Both are available from the ODBC Administrator and implemented by the Driver Manager. Tracing provides a means to record to a log file the sequence of function calls. Visual Studio Analyzer allows the analysis of information about the interaction of components in a distributed environment.

This section contains the following topics.

- Tracing
- Visual Studio Analyzer

Tracing
The ODBC Driver Manager has a trace facility that allows the sequence of function calls made by an ODBC application to be recorded and transcribed into a log file. Tracing is performed by a trace DLL that captures calls between the application and the Driver Manager, and between the Driver Manager and the driver. This method of tracing replaces the tracing performed by the ODBC 2.x Driver Manager and the tracing performed in ODBC 2.x by ODBC Spy.

This section contains the following topics.

- **Trace DLL**
- **Trace File**
- **Enabling Tracing**
- **Dynamic Tracing**

### Trace DLL

The DLL that performs tracing is one of the ODBC core components. The trace DLL is currently provided as a sample DLL in the ODBC component of the Windows SDK, and was formerly included the Microsoft Data Access Components (MDAC) SDK. Therefore, the registry entry, interface, and sample code for the trace DLL are available. This DLL can be replaced by a trace DLL produced by either an ODBC user or a third-party vendor. A custom trace DLL should be given a different name than the original sample trace DLL. Trace DLLs must be installed in the system directory, or they will fail to load. The connection strings will not be passed to the trace DLL by the Driver Manager.

The trace DLL traces input arguments, output arguments, deferred arguments, return codes, and SQLSTATEs. When tracing is enabled, the Driver Manager calls the trace DLL at two points: once upon function entry (before argument validation) and again just before the function returns.

When an application calls a function, the Driver Manager calls a trace function in the trace DLL before calling the function in the driver or processing the call itself. Each ODBC function has a corresponding trace function (prefixed with `Trace`) that is identical to the ODBC function with the exception of the name. When the trace function is called, the trace DLL captures the input arguments and returns a return code. Because the trace DLL is called before the Driver Manager validates arguments, invalid function calls are traced, so state transition errors and invalid arguments are logged.

After calling the trace function in the trace DLL, the Driver Manager calls the ODBC function in the driver. It then calls `TraceReturn` in the trace DLL. This function takes two arguments: the value returned by the trace DLL for the trace function, and the return code returned by the driver to the Driver Manager for the ODBC function (or the value returned by the Driver Manager itself if it processed the function). The function uses the value returned for the trace function to manipulate captured input argument values. It writes the code returned for the ODBC function to the log file (or displays it dynamically, if that is enabled). It dereferences the output argument pointers and logs the output argument values.

### Trace File

An application specifies the trace file either by setting the `TraceFile` keyword in the Odbc.ini registry entry or by calling `SQLSetConnectAttr` with the SQL_ATTR_TRACEFILE connection attribute. If the file does not exist when tracing is enabled, the Driver Manager will create the file. Each application should have its own dedicated trace file to avoid contention. An application can use more than one trace file; an application's setup program can provide the user with a choice of trace files. If tracing is enabled dynamically, an application can also display trace results, rather
than logging to the trace file.

The trace file provides a log of each ODBC function call with the data types and values of all arguments. It logs all input functions and logs all returned functions with return codes and error states.

In ODBC 3.x, parameters to connection functions are not provided to the trace DLL.

© 2016 Microsoft

Enabling Tracing

Tracing can be enabled in the following three ways:

- Set the Trace and TraceFile keywords in the Odbc.ini registry entry. This enables or disables tracing when SQLAllocHandle with a HandleType of SQL_HANDLE_ENV is called. These options are set in the Tracing tab of the ODBC Data Source Administrator dialog box displayed during data source setup. For more information, see Registry Entries for Data Sources.

- Call SQLSetConnectAttr to set the SQL_ATTR_TRACE connection attribute to SQL_OPT_TRACE_ON. This enables or disables tracing for the duration of the connection. For more information, see the SQLSetConnectAttr function description.

- Use ODBCSharedTraceFlag to turn tracing on or off dynamically. (For more information, see the next topic, Dynamic Tracing.)

© 2016 Microsoft

Dynamic Tracing

Tracing can be enabled or disabled at any point in an application run. This allows an application to trace any number of function calls.

The variable ODBCSharedTraceFlag is set to enable tracing dynamically. This variable is shared among all running copies of the Driver Manager. If any application sets this variable, tracing is enabled for all ODBC applications currently running. To turn tracing off when dynamic tracing is enabled, an application calls SQLSetConnectAttr to set SQL_ATTR_TRACE to SQL_TRACE_OFF. This call will turn tracing off for that application only. Applications that are linked with Odbc32.lib can modify use of this variable. Trace data can be displayed in a real-time window, instead of the trace file, which must be opened after the ODBC session. Controls can be added to an application's screen to turn tracing on or off at will.

The trace DLL shipped with ODBC 3.x is not thread-safe. It is not guaranteed that the log file is written correctly if global tracing is enabled (the variable ODBCSharedTraceFlag is set) and more than one application writes to the trace file at the same time. This condition does not return an error.

© 2016 Microsoft

Visual Studio Analyzer

Important

Support for Visual Studio Analyzer was removed beginning in Windows 8 (Visual Studio Analyzer was only
Microsoft® Visual Studio™ Analyzer provides a high-level view of the ODBC application. The developer can evaluate, analyze, and debug the structure, performance, and interactions from the application’s perspective, rather than from a component or code perspective. The information gathered by Visual Studio Analyzer is in the form of events, which represent some kind of interaction between two components of the application. Events can be composed of function calls and returns from functions, such as connections, database queries, and transactions.

This section contains the following topic.

- **Enabling Visual Studio Analyzer**

© 2016 Microsoft

## Enabling Visual Studio Analyzer

**Important**

Support for Visual Studio Analyzer was removed beginning in Windows 8 (Visual Studio Analyzer was only included in older versions of Visual Studio.). For an alternative troubleshooting mechanism, use BID tracing.

Microsoft Visual Studio Analyzer is an application-analysis tool designed to provide a high-level look at an application's performance across all tiers and systems. It focuses on the interaction between components. Visual Studio Analyzer can be started and stopped from the ODBC Administrator’s Tracing tab.

To view any of the events that ODBC generates, follow these steps:

1. Access the ODBC Data Source Administrator.
2. Click the **Tracing** tab.
3. Click **Start Visual Studio Analyzer Tracing**.
4. Click **OK**.

Visual Studio Analyzer event-generation continues until **Stop Visual Studio Analyzer Tracing** is selected.

This section contains the following topic.

- **Events Generated by the ODBC Driver Manager**

© 2016 Microsoft

## Events Generated by the ODBC Driver Manager

**Important**

Support for Visual Studio Analyzer was removed beginning in Windows 8 (Visual Studio Analyzer was only included in older versions of Visual Studio.). For an alternative troubleshooting mechanism, use BID tracing.
Events generated by the ODBC Driver Manager are registered when the Start Visual Studio Analyzer button is clicked. The tool itself offers system-defined events and the ability to create custom events. For more information about events, see the Visual Studio Analyzer Reference Guide within the Visual Studio suite of documentation.

<table>
<thead>
<tr>
<th>Visual Studio Analyzer event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>Generated on every ODBC API entry.</td>
</tr>
<tr>
<td>ReturnException</td>
<td>Generated on every ODBC API return if the return code is SQL_ERROR.</td>
</tr>
<tr>
<td>ReturnNormal</td>
<td>Generated on every ODBC API return if the return code is not SQL_ERROR.</td>
</tr>
<tr>
<td>Connection Start</td>
<td>Indicates that a connection started; generated when the ODBC Driver Manager calls the driver's connection APIs.</td>
</tr>
<tr>
<td>Connection Complete</td>
<td>Indicates that a connection completed; generated when the driver's connection APIs return to the ODBC Driver Manager.</td>
</tr>
<tr>
<td>Disconnect Start</td>
<td>Generated when the ODBC Driver Manager calls the driver's SQLDisconnect function.</td>
</tr>
<tr>
<td>Disconnect Complete</td>
<td>Generated when the driver's SQLDisconnect function returns to the ODBC Driver Manager.</td>
</tr>
<tr>
<td>QuerySend</td>
<td>Generated when the ODBC Driver Manager calls the driver's SQLPrepare, SQLExecute, SQLExecDirect functions, as well as catalog functions such as SQLTables and SQLColumns.</td>
</tr>
<tr>
<td>QueryResult</td>
<td>Generated when the driver returns a result set to the ODBC Driver Manager for functions involving queries.</td>
</tr>
<tr>
<td>TransactionStart</td>
<td>Generated when an application sets the value of SQL_ATTR_AUTOCOMMIT to SQL_AUTOCOMMIT_OFF, or after an application successfully calls SQLEndTran.</td>
</tr>
<tr>
<td>TransactionCommit</td>
<td>Generated when an application calls SQLEndTran to commit a local transaction.</td>
</tr>
<tr>
<td>TransactionRollback</td>
<td>Generated when an application calls SQLEndTran to roll back a local transaction.</td>
</tr>
<tr>
<td>JoinDTC</td>
<td>Generated when an application joins the Distributed Transaction Coordinator (DTC).</td>
</tr>
<tr>
<td>LeaveDTC</td>
<td>Generated when an application leaves the Distributed Transaction Coordinator (DTC).</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

Driver–Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes

Drivers can allocate driver-specific values for the following:
- **SQL Data Type Indicators** These are used in `ParameterType` in `SQLBindParameter` and in `DataType` in `SQLGetTypeInfo` and returned by `SQLColAttribute`, `SQLColumns`, `SQLDescribeCol`, `SQLGetTypeInfo`, `SQLDescribeParam`, `SQLProcedureColumns`, and `SQLSpecialColumns`.

- **Descriptor Fields** These are used in `FieldIdentifier` in `SQLColAttribute`, `SQLGetDescField`, and `SQLSetDescField`.

- **Diagnostic Fields** These are used in `DiagIdentifier` in `SQLGetDiagField` and `SQLGetDiagRec`.

- **Information Types** These are used in `InfoType` in `SQLGetInfo`.

- **Connection and Statement Attributes** These are used in `Attribute` in `SQLGetConnectAttr`, `SQLGetStmtAttr`, `SQLSetConnectAttr`, and `SQLSetStmtAttr`.

For each of these items, there are two sets of values: values reserved for use by ODBC, and values reserved for use by drivers. Before implementing driver-specific values, a driver writer must request a value for each driver-specific type, field, or attribute from Open Group. For new driver development, use the range described in the table below. The ODBC 3.8 Driver Manager will not generate an error if an unknown value is used that is not in the range described below. However, later versions of the Driver Manager might generate an error if unknown values are received that are not in the range.

When any of these values is passed to an ODBC function, the driver must check whether the value is valid. Drivers return SQLSTATE HYC00 (Optional feature not implemented) for driver-specific values that apply to other drivers.

Starting with ODBC 3.8, driver writers can allocate driver-specific attributes within a reserved range.

<table>
<thead>
<tr>
<th>Attribute type</th>
<th>ODBC data type</th>
<th>Driver-specific range base</th>
<th>Driver-specific range limit</th>
<th>ODBC constant for driver-specific value range base</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL data type indicators</td>
<td>SQLSMALLINT</td>
<td>0x4000</td>
<td>0x7FFF</td>
<td>SQL_DRIVER_SQL_TYPE_BASE</td>
</tr>
<tr>
<td>Descriptor fields</td>
<td>SQLSMALLINT</td>
<td>0x4000</td>
<td>0x7FFF</td>
<td>SQL_DRIVER_DESCRIPTOR_BASE</td>
</tr>
<tr>
<td>Diagnostic fields</td>
<td>SQLSMALLINT</td>
<td>0x4000</td>
<td>0x7FFF</td>
<td>SQL_DRIVER_DIAGNOSTIC_BASE</td>
</tr>
<tr>
<td>Information types</td>
<td>SQLUSMALLINT</td>
<td>0x4000</td>
<td>0x7FFF</td>
<td>SQL_DRIVER_INFO_TYPE_BASE</td>
</tr>
<tr>
<td>Connection attributes</td>
<td>SQLINTEGER</td>
<td>0x00004000</td>
<td>0x00007FFF</td>
<td>SQL_DRIVER_CONNECT_ATTR_BASE</td>
</tr>
<tr>
<td>Statement attributes</td>
<td>SQLINTEGER</td>
<td>0x00004000</td>
<td>0x00007FFF</td>
<td>SQL_DRIVER_STATEMENT_ATTR_BASE</td>
</tr>
</tbody>
</table>

**Note**

The ODBC 3.8 Driver Manager neither validates nor enforces these ranges for backward compatibility. A future version of the Driver Manager might enforce them, however.

**Note**

Driver-specific data types, descriptor fields, diagnostic fields, information types, statement attributes, and connection attributes must be described in the driver documentation. When any of these values is passed to an ODBC function, the driver must check whether the value is valid. Drivers return SQLSTATE HYC00 (Optional...
feature not implemented) for driver-specific values that apply to other drivers.
The base values are defined to facilitate driver development. For example, driver specific diagnostic attributes can be defined in the following format:

```
SQL_DRIVER_DIAGNOSTIC_BASE+0, SQL_DRIVER_DIAGNOSTIC_BASE +1
```

Backward Compatibility and Standards Compliance

Backward compatibility is the ability of newer ODBC components to work with old ODBC components. The following sections discuss how these components are affected by the changes in ODBC 3.x. The information contained in them primarily addresses the writing of an ODBC 3.x application and how backward compatibility issues are handled by ODBC drivers. For specific guidelines about how backward compatibility issues affect the writing of an ODBC 3.x driver, see Appendix G: Driver Guidelines for Backward Compatibility.

This section contains the following topics.

- Affected ODBC Components
- Types of Changes
- Application/Driver Compatibility
- New Features
- Duplicated Features
- Behavioral Changes
- Writing ODBC 3.x Applications
- Writing ODBC 3.x Drivers

Affected ODBC Components

Backward compatibility describes how applications, the Driver Manager, and drivers are affected by the introduction of a new version of the Driver Manager. This affects applications and driver when either or both of them remain in the old version. There are, therefore, three types of backward compatibility to consider, as shown in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Version of DM</th>
<th>Version of application</th>
<th>Version of driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backward Compatibility of Driver Manager</td>
<td>3.x</td>
<td>2.x</td>
<td>2.x</td>
</tr>
<tr>
<td>Backward Compatibility of Driver[1]</td>
<td>3.x</td>
<td>2.x</td>
<td>3.x</td>
</tr>
</tbody>
</table>
The backward compatibility of drivers is primarily discussed in Appendix G: Driver Guidelines for Backward Compatibility.

**Note**

A standards-compliant application — for example, an application that has been written in accordance with the Open Group or ISO CLI standards — is guaranteed to work with an ODBC 3.x driver through the ODBC 3.x Driver Manager. It is assumed that the functionality that the application is using is available in the driver. It is also assumed that the standards-compliant application has been compiled with the ODBC 3.x header files.

### Types of Changes

Three types of changes are made in ODBC 3.x (and any version of ODBC). Each of these affects backward compatibility differently and is handled in a different way. These changes are described in the following table.

<table>
<thead>
<tr>
<th>Type of change</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New features</td>
<td>These are features that are new to ODBC 3.x, such as out-of-line binding or descriptors. These are implemented only when the application and driver, as well as the Driver Manager, are of version 3.x, so there is no attempt to make these backward compatible.</td>
</tr>
<tr>
<td>Duplicated features</td>
<td>These are features that exist in ODBC 2.x and ODBC 3.x but are implemented in different ways in each. The functions <code>SQLAllocHandle</code> and <code>SQLAllocStmt</code> are an example. Backward compatibility issues for these and other duplicated features are mostly handled by mappings in the Driver Manager.</td>
</tr>
<tr>
<td>Behavioral changes</td>
<td>These are features that are handled differently in ODBC 2.x and ODBC 3.x. A datetime <code>#define</code> is an example. These features are handled by the ODBC 3.x driver based on an environment attribute setting. (See Behavioral Changes for more information.)</td>
</tr>
</tbody>
</table>

### Application/Driver Compatibility

ODBC applications and driver fall into a number of categories in addition to their version. Some of these applications are incompatible with some drivers; in other cases, the type of the application or driver may have a bearing on the backward compatibility issues between them.

This section contains the following topics.

- Types of Applications
- Types of Drivers
- Compatibility Matrix
Types of Applications

ODBC applications can be classified as follows:

- **Pure ODBC 2.x Application** A 32-bit application that:
  - Calls only ODBC 2.x functions (including the ODBC 1.0 function `SQLSetParam`). These include ODBC 1.x applications that have been ported to 32-bit.
  - Expects ODBC 2.x behavior for features that have had behavioral changes. (See Behavioral Changes for more information.)
  - Has not been recompiled with ODBC 3.5 headers.

- **Pure ODBC 2.x Recompiled Application** A pure ODBC 2.x application that has been recompiled using the ODBC 3.5 header files, by setting `ODBCVER=0x0250`.

- **Pure ODBC 2.x Unicode Application** A pure ODBC 2.x recompiled application that is Unicode-compliant and uses the SQL_WCHAR data type.

- **Pure Open Group and ISO–compliant ODBC Application** A 32-bit application that:
  - Calls functions defined in the Open Group or ISO CLI standards. (These functions may include deprecated 3.0 functions.)
  - Does not use the Unicode data types.
  - Expects ODBC 3.0 behavior for features that have had behavioral changes.

- **Pure ODBC 3.0 Application** A 32-bit application that:
  - Is compiled with 3.0 headers.
  - Calls any ODBC 3.0 function, possibly including those that are deprecated.
  - Expects ODBC 3.0 behavior for features that have had behavioral changes.

- **Pure ODBC 3.5 Application** A 32 or 64-bit application that:
  - May use Unicode data types.
  - Calls any ODBC 3.5 function, possibly including those that are deprecated.
  - Expects ODBC 3.5 behavior for features that have had behavioral changes.

- **Pure ODBC 3.8 (or later) Application** A 32-bit or 64-bit application that:
  - May use Unicode data types.
  - Calls any ODBC 3.8 function, possibly including those that are deprecated.
  - Expects ODBC 3.8 behavior for features that have had behavioral changes.

- **Replaced Application** A 32 or 64-bit application that:
  - Implements new behavior for duplicated functionality.
  - Uses any new features in a later version of ODBC only within conditional code.
  - Has limited conditional code to handle behavioral changes or has registered itself to be an earlier
Types of Drivers

ODBC drivers can be classified as follows:

- **32-bit ODBC 2.x Driver** A 32-bit driver that:
  - Exports only ODBC 2.x functions.
  - Exhibits ODBC 2.x behavior for behavioral changes.

- **ISO and Open Group-Compliant Driver** A 32-bit driver that:
  - Exports all functions that are documented in the Open Group or ISO CLI documents. This will include some of functions that are deprecated in ODBC.
  - Exhibits ODBC 3.0 behavior for behavioral changes.
  - Does not necessarily go through the ODBC 3.0 Driver Manager.

- **ODBC 3.0 Driver** A 32-bit driver that:
  - Exports only functions that are in ODBC 3.0 minus deprecated functions.
  - Is capable of exhibiting ODBC 2.x behavior or ODBC 3.0 behavior with respect to behavioral changes, based on the SQL\_ATTR\_APP\_ODBC\_VERSION environment attribute.

- **ODBC 3.5 (or later) ANSI Driver** A 32-bit driver that:
  - Exports only functions that are in ODBC 3.5 minus deprecated functions.
  - Is capable of exhibiting ODBC 2.x behavior or ODBC 3.0 behavior, or ODBC 3.5 behavior with respect to behavioral changes, based on the SQL\_ATTR\_APP\_ODBC\_VERSION environment attribute.

- **ODBC 3.5 (or later) Unicode Driver** A 32-bit driver that:
  - Supports all the features of an ODBC 3.5 ANSI driver.
  - Exports Unicode versions of all ODBC string APIs.
  - Can store and process Unicode data on the data source.

**Note**

16-bit ODBC drivers will not work directly with the ODBC 3.x Driver Manager. However, it is possible for 16-bit drivers to work with the 2.0 ODBC Driver Manager, which subsequently thunks up to the 3.x Driver Manager.

Compatibility Matrix

The following table describes the compatibility of the types of applications and drivers defined previously in this section.
<table>
<thead>
<tr>
<th>Application type and version</th>
<th>32-bit ODBC 2.x driver</th>
<th>ODBC 3.x driver</th>
<th>ODBC 3.8 driver</th>
<th>ISO and Open Group–compliant driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit application, any version</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Compatible</td>
</tr>
<tr>
<td>Pure 2.x application</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Not compatible[3]</td>
</tr>
<tr>
<td>Pure 3.0 application</td>
<td>Not compatible</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Not compatible[4]</td>
</tr>
<tr>
<td>Pure 3.5 application</td>
<td>Not compatible</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Not compatible[4]</td>
</tr>
<tr>
<td>Replaced application</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Compatible</td>
<td>Not compatible[3]</td>
</tr>
</tbody>
</table>

[1] The application must recompile using ODBC 3.5 (or higher) headers with the UNICODE option (if it is a Unicode application) and must set ODBCVER to 0x0250.

[2] The application must compile using ODBC 3.5 (or higher) headers and link with the ODBC Driver Manager. It must also set the header flag ODBC_STD.

[3] This configuration can potentially fail to work because there are features in ODBC 2.x that are not in the standards, such as bookmarks.

[4] This configuration can potentially fail to work because there are features in ODBC 3.x that are not in the standards, such as bookmarks.

[5] This configuration can potentially fail because there are features in ODBC 3.8 that are not in ODBC 2.x or 3.x drivers, such as driver-specific C Data Types in ODBC.

**Driver Manager Compatibility**

An ODBC 3.0 application that must operate with all Driver Manager versions should do the following on startup:

- Allocate an environment handle.
- Set the SQL_ATTR_ODBC_VERSION environment attribute to SQL_OV_ODBC3_80. If the Driver Manager returns SQL_ERROR, the Driver Manager is older than 3.8. Reset SQL_ATTR_ODBC_VERSION to SQL_OV_ODBC3 or SQL_OV_ODBC2, as appropriate, to correspond to the Driver Manager.
- Allocate a connection handle.
- Make a connection.
• Call SQLGetInfo for SQL_DRIVER_ODBC_VER to determine the driver version. If the driver is an ODBC 3.8
driver, you can use driver-specific C types. Otherwise, do not use driver-specific C data types.

Note that a recompiled ODBC 3.x application can use ODBC 3.8 features other than driver-specific C types without
specifying SQL_OV_ODBC3_80 for SQL_ATTR_ODBC_VERSION. This is similar to a recompiled ODBC 2.x
application using ODBC 3.x features.

Using SQLCancelHandle in an Application Compatible with all Driver Managers

Because SQLCancelHandle Function is not supported in Driver Managers that were released before Windows 7, an
application cannot be loaded in older versions of Windows if it calls SQLCancelHandle directly. To work with all
versions of Driver Managers and use SQLCancelHandle on new versions of Windows, an application should call
SQLCancelHandle indirectly by using LoadLibrary and GetProcAddress.

See Also

What's New in ODBC 3.8

© 2016 Microsoft

New Features

The following new functionality has been introduced in ODBC 3.x. An ODBC 3.x application working with an ODBC
2.x driver will not be able to use this functionality. The ODBC 3.x Driver Manager does not map these features when
working with an ODBC 2.x driver.

• Functions that take a descriptor handle as an argument: SQLSetDescField, SQLGetDescField,
  SQLSetDescRec, SQLGetDescRec, and SQLCopyDesc.

• The functions SQLSetEnvAttr and SQLGetEnvAttr.

• The use of SQLAllocHandle to allocate a descriptor handle. (The use of SQLAllocHandle to allocate
  environment, connection, and statement handles is duplicated, not new, functionality.)

• The use of SQLGetConnectAttr to get the SQL_ATTR_AUTO_IPD connection attributes. (The use of
  SQLSetConnectAttr to set, and SQLGetConnectAttr to get, other connection attributes is duplicated, not
  new, functionality.)

• The use of SQLSetStmtAttr to set, and SQLGetStmtAttr to get, the following statement attributes. (The use
  of SQLSetStmtAttr to set, and SQLGetStmtAttr to get, other statement attributes is duplicated, not new,
  functionality.)

  SQL_ATTR_APP_ROW_DESC
  SQL_ATTR_APP_PARAM_DESC
  SQL_ATTR_ENABLE_AUTO_IPD
  SQL_ATTR_FETCH_BOOKMARK_PTR
The use of `SQLGetStmtAttr` to get the following statement attributes. (The use of `SQLGetStmtAttr` to get other statement attributes is duplicated functionality, not new functionality.)

- `SQL_ATTR_IMP_ROW_DESC` `SQL_ATTR_IMP_PARAM_DESC`

- Use of the interval C data type, the interval SQL data types, the BIGINT C data types, and the `SQL_C_NUMERIC` data structure.

- Row-wise binding of parameters.

- Offset-based bookmark fetches, such as calling `SQLFetchScroll` with a `FetchOrientation` argument of `SQL_FETCH_BOOKMARK` and specifying an offset other than 0.

- `SQLFetch` returning the row status array, number of rows fetched, fetching multiple rows, intermixing calls with `SQLFetchScroll`, and intermixing calls with `SQLBulkOperations` or `SQLSetPos`. For more information, see the next section, Block Cursors, Scrollable Cursors, and Backward Compatibility for ODBC 3.x Applications.

- Named parameters.

- Any of the ODBC 3.x-specific `SQLGetInfo` options. (If an ODBC 3.x application working with an ODBC 2.x driver calls the `SQL_XXX_CURSOR_ATTRIBUTES1` information types, which have replaced several ODBC 2.x information types, some of the information might be reliable, but some might be unreliable. For more information, see `SQLGetInfo`.)

- Bind offsets.

- Updating, refreshing, and deleting by bookmarks (through a call to `SQLBulkOperations`).

- Calling `SQLBulkOperations` or `SQLSetPos` in the S5 state.

- The `ROW_NUMBER` and `COLUMN_NUMBER` fields in the diagnostic record (which have to be retrieved by the replacement functions `SQLGetDiagField` or `SQLGetDiagRec`).

- Approximate row counts.

- Warning information (`SQL_ROW_SUCCESS_WITH_INFO` from `SQLFetchScroll`).

- Variable-length bookmarks.

- Extended error information for arrays of parameters.

- All of the new columns in the result sets returned by the catalog functions.
• Use of `SQLDescribeCol` and `SQLColAttribute` on column 0.
• Use of any ODBC 3.x–specific column attributes in a call to `SQLColAttribute`.
• Use of multiple environment handles.

This section contains the following topic.

• **Block Cursors, Scrollable Cursors, and Backward Compatibility for ODBC 3.x Applications**

© 2016 Microsoft

**Block Cursors, Scrollable Cursors, and Backward Compatibility for ODBC 3.x Applications**

The existence of both `SQLFetchScroll` and `SQLExtendedFetch` represents the first clear split in ODBC between the Application Programming Interface (API), which is the set of functions the application calls, and the Service Provider Interface (SPI), which is the set of functions the driver implements. This split is required to balance the requirement in ODBC 3.x, which uses `SQLFetchScroll`, to align with the standards and be compatible with ODBC 2.x, which uses `SQLExtendedFetch`.

The ODBC 3.x API, which is the set of functions the application calls, includes `SQLFetchScroll` and related statement attributes. The ODBC 3.x SPI, which is the set of functions the driver implements, includes `SQLFetchScroll`, `SQLExtendedFetch`, and related statement attributes. Because ODBC does not formally enforce this split between the API and the SPI, it is possible for ODBC 3.x applications to call `SQLExtendedFetch` and related statement attributes. However, there is no reason for ODBC 3.x applications to do this. For more information about APIs and SPIs, see the introduction to **ODBC Architecture**.

For information about how the ODBC 3.x Driver Manager maps calls to ODBC 2.x and ODBC 3.x drivers, and what functions and statement attributes an ODBC 3.x driver should implement for block and scrollable cursors, see **What the Driver Does** in Appendix G: Driver Guidelines for Backward Compatibility.

The following table summarizes what functions and statement attributes an ODBC 3.x application should use with block and scrollable cursors. It also lists changes between ODBC 2.x and ODBC 3.x in this area that ODBC 3.x applications should be aware of to be compatible with ODBC 2.x drivers.

<table>
<thead>
<tr>
<th>Function or statement attribute</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SQL_ATTR_FETCH_BOOKMARK_PTR</code></td>
<td>Points to the bookmark to use with <code>SQLFetchScroll</code>. When an application sets this in an ODBC 2.x driver, this must point to a fixed-length bookmark.</td>
</tr>
<tr>
<td><code>SQL_ATTR_ROW_STATUS_PTR</code></td>
<td>Points to the row status array filled by <code>SQLFetch</code>, <code>SQLFetchScroll</code>, <code>SQLBulkOperations</code>, and <code>SQLSetPos</code>. If an application sets this in an ODBC 2.x driver and calls <code>SQLBulkOperation</code> with an Operation of <code>SQL_ADD</code> before calling <code>SQLFetchScroll</code>, <code>SQLFetch</code>, or <code>SQLExtendedFetch</code>, SQLSTATE HY011 (Attribute cannot be set now) is returned.</td>
</tr>
</tbody>
</table>
When an application calls `SQLFetch` in an ODBC 2.x driver, `SQLFetch` is mapped to `SQLExtendedFetch` and therefore returns values in this array.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_ROWS_FETCHED_PTR</td>
<td>Points to the buffer in which <code>SQLFetch</code> and <code>SQLFetchScroll</code> return the number of rows fetched. When an application calls <code>SQLFetch</code> in an ODBC 2.x driver, <code>SQLFetch</code> is mapped to <code>SQLExtendedFetch</code> and therefore returns a value in this buffer.</td>
</tr>
<tr>
<td>SQL_ATTR_ROW_ARRAY_SIZE</td>
<td>Sets the rowset size. If an application calls <code>SQLBulkOperations</code> with an <code>Operation</code> of SQL_ADD in an ODBC 2.x driver, SQL_ROWSET_SIZE will be used for the call, not SQL_ATTR_ROW_ARRAY_SIZE, because the call is mapped to <code>SQLSetPos</code> with an <code>Operation</code> of SQL_ADD, which uses SQL_ROWSET_SIZE. Calling <code>SQLSetPos</code> with an <code>Operation</code> of SQL_ADD or <code>SQLExtendedFetch</code> in an ODBC 2.x driver uses SQL_ROWSET_SIZE. Calling <code>SQLFetch</code> or <code>SQLFetchScroll</code> in an ODBC 2.x driver uses SQL_ATTR_ROW_ARRAY_SIZE.</td>
</tr>
</tbody>
</table>

**SQLBulkOperations**

Performs insert and bookmark operations. When `SQLBulkOperations` with an `Operation` of SQL_ADD is called in an ODBC 2.x driver, it is mapped to `SQLSetPos` with an `Operation` of SQL_ADD. The following are implementation details:

- When working with an ODBC 2.x driver, an application must use only the implicitly allocated ARD associated with the `StatementHandle`; it cannot allocate another ARD for adding rows, because explicit descriptor operations are not supported in an ODBC 2.x driver. An application must use `SQLBindCol` to bind to the ARD, not `SQLSetDescField` or `SQLSetDescRec`.
- When calling an ODBC 3.x driver, an application can call `SQLBulkOperations` with an `Operation` of SQL_ADD before calling `SQLFetch` or `SQLFetchScroll`. When calling an ODBC 2.x driver, an application must call `SQLFetchScroll` before calling `SQLBulkOperations` with an Operation of SQL_ADD.

**SQLFetch**

Returns the next rowset. The following are implementation details:

- When an application calls `SQLFetch` in an ODBC 2.x driver, it is mapped to `SQLExtendedFetch`.
- When an application calls `SQLFetch` in an ODBC 3.x driver, it returns the number of rows specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.

**SQLFetchScroll**

Returns the specified rowset. The following are implementation details:

- When an application calls `SQLFetchScroll` in an ODBC 2.x driver, it returns SQLSTATE 01S01 (Error in row) before each error that applies to a single row. It does this only because the ODBC 3.x Driver Manager maps this to `SQLExtendedFetch`. When an application calls `SQLFetch` in an ODBC 2.x driver, `SQLFetch` is mapped to `SQLExtendedFetch` and therefore returns values in this array.
and `SQLExtendedFetch` returns this SQLSTATE. When an application calls `SQLFetchScroll` in an ODBC 3.x driver, it never returns SQLSTATE 01S01 (Error in row).

- When an application calls `SQLFetchScroll` in an ODBC 2.x driver with `FetchOrientation` set to SQL_FETCH_BOOKMARK, the `FetchOffset` argument must be set to 0. SQLSTATE HYC00 (Optional feature not implemented) is returned if offset-based bookmark fetching is attempted with an ODBC 2.x driver.

**Note**

ODBC 3.x applications should not use `SQLExtendedFetch` or the SQL_ROWSET_SIZE statement attribute. Instead, they should use `SQLFetchScroll` and the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. ODBC 3.x applications should not use `SQLSetPos` with an `Operation` of SQL_ADD but should use `SQLBulkOperations` with an `Operation` of SQL_ADD.

---

**Duplicated Features**

The following ODBC 2.x functions have been duplicated by ODBC 3.x functions. As a result, the ODBC 2.x functions are deprecated in ODBC 3.x. The ODBC 3.x functions are referred to as replacement functions.

When an application uses a deprecated ODBC 2.x function and the underlying driver is an ODBC 3.x driver, the Driver Manager maps the function call to the corresponding replacement function. The only exception to this rule is `SQLExtendedFetch`. (See the footnote at the end of the following table.) For more information about these mappings, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.

When an application uses a replacement function and the underlying driver is an ODBC 2.x driver, the Driver Manager maps the function call to the corresponding deprecated function.

<table>
<thead>
<tr>
<th>ODBC 2.x function</th>
<th>ODBC 3.x function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocConnect</td>
<td>SQLAllocHandle</td>
</tr>
<tr>
<td>SQLAllocEnv</td>
<td>SQLAllocHandle</td>
</tr>
<tr>
<td>SQLAllocStmt</td>
<td>SQLAllocHandle</td>
</tr>
<tr>
<td>SQLColAttributes</td>
<td>SQLColAttribute</td>
</tr>
<tr>
<td>SQLError</td>
<td>SQLGetDiagRec</td>
</tr>
<tr>
<td>SQLExtendedFetch[1]</td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td>SQLFreeConnect</td>
<td>SQLFreeHandle</td>
</tr>
<tr>
<td>SQLFreeEnv</td>
<td>SQLFreeHandle</td>
</tr>
<tr>
<td>SQLGetConnectOption</td>
<td>SQLGetConnectAttr</td>
</tr>
<tr>
<td>SQLGetStmtOption</td>
<td>SQLGetStmtAttr</td>
</tr>
</tbody>
</table>

[1] See the footnote at the end of this table.
The function `SQLExtendedFetch` is duplicated functionality; `SQLFetchScroll` provides the same functionality in ODBC 3.x. However, the Driver Manager does not map `SQLExtendedFetch` to `SQLFetchScroll` when going against an ODBC 3.x driver. For more information, see What the Driver Manager Does in Appendix G: Driver Guidelines for Backward Compatibility. The Driver Manager maps `SQLFetchScroll` to `SQLExtendedFetch` when going against an ODBC 2.x driver.

Note

The function `SQLBindParam` is a special case. `SQLBindParam` is duplicated functionality. This is not an ODBC 2.x function, but a function that is present in the Open Group and ISO standards. The functionality provided by this function is completely subsumed by that of `SQLBindParameter`. As a result, the Driver Manager maps a call to `SQLBindParam` to `SQLBindParameter` when the underlying driver is an ODBC 3.x driver. However, when the underlying driver is an ODBC 2.x driver, the Driver Manager does not perform this mapping.

© 2016 Microsoft

Behavioral Changes

Behavioral changes are those changes for which the syntax of the interface remains the same, but the semantics have changed. For these changes, functionality used in ODBC 2.x behaves differently than the same functionality in ODBC 3.x.

Whether an application exhibits ODBC 2.x behavior or ODBC 3.x behavior is determined by the `SQL_ATTR_ODBC_VERSION` environment attribute. This 32-bit value is set to SQL_OV_ODBC2 to exhibit ODBC 2.x behavior, and SQL_OV_ODBC3 to exhibit ODBC 3.x behavior.

The `SQL_ATTR_ODBC_VERSION` environment attribute is set by a call to `SQLSetEnvAttr`. After an application calls `SQLAllocHandle` to allocate an environment handle, it must call `SQLSetEnvAttr` immediately to set the behavior it exhibits. (As a result, there is a new environment state to describe the environment handle in an allocated, but versionless, state.) For more information, see Appendix B: ODBC State Transition Tables.

An application states what behavior it exhibits with the `SQL_ATTR_ODBC_VERSION` environment attribute, but the attribute has no effect on the application's connection with an ODBC 2.x or ODBC 3.x driver. An ODBC 3.x application can connect to either an ODBC 2.x or 3.x driver, no matter what the setting of the environment attribute.

ODBC 3.x applications should never call `SQLAllocEnv`. As a result, if the Driver Manager receives a call to `SQLAllocEnv`, it recognizes the application as an ODBC 2.x application.

The `SQL_ATTR_ODBC_VERSION` attribute affects three different aspects of an ODBC 3.x driver’s behavior:

- SQLSTATEs
- Data types for date, time, and timestamp
- The `CatalogName` argument in `SQLTables` accepts search patterns in ODBC 3.x, but not in ODBC 2.x

The setting of the `SQL_ATTR_ODBC_VERSION` environment attribute does not affect `SQLSetParam` or
SQLBindParam. SQLColAttribute is also not affected by this bit. Although SQLColAttribute returns attributes that are affected by the version of ODBC (date type, precision, scale and length), the intended behavior is determined by the value of the FieldIdentifier argument. When FieldIdentifier is equal to SQL_DESC_TYPE, SQLColAttribute returns the ODBC 3.x codes for date, time, and timestamp; when FieldIdentifier is equal to SQL_COLUMN_TYPE, SQLColAttribute returns the ODBC 2.x codes for date, time, and timestamp.

This section contains the following topics.

- SQLSTATE Mappings
- Datetime Data Type Changes

© 2016 Microsoft

SQLSTATE Mappings

This topic discusses SQLSTATE values for ODBC 2.x and ODBC 3.x. For more information on ODBC 3.x SQLSTATE values, see Appendix A: ODBC Error Codes.

In ODBC 3.x, HYxxx SQLSTATEs are returned instead of S1xxx, and 42Sxx SQLSTATEs are returned instead of S00XX. This was done to align with Open Group and ISO standards. In many cases, the mapping is not one-to-one because the standards have redefined the interpretation of several SQLSTATEs.

When an ODBC 2.x application is upgraded to an ODBC 3.x application, the application has to be changed to expect ODBC 3.x SQLSTATEs instead of ODBC 2.x SQLSTATEs. The following table lists the ODBC 3.x SQLSTATEs that each ODBC 2.x SQLSTATE is mapped to.

When the SQL_ATTR_ODBC_VERSION environment attribute is set to SQL_OV_ODBC2, the driver posts ODBC 2.x SQLSTATEs instead of ODBC 3.x SQLSTATEs when SQLGetDiagField or SQLGetDiagRec is called. A specific mapping can be determined by noting the ODBC 2.x SQLSTATE in column 1 of the following table that corresponds to the ODBC 3.x SQLSTATE in column 2.

<table>
<thead>
<tr>
<th>ODBC 2.x SQLSTATE</th>
<th>ODBC 3.x SQLSTATE</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>01S03</td>
<td>01001</td>
<td></td>
</tr>
<tr>
<td>01S04</td>
<td>01001</td>
<td></td>
</tr>
<tr>
<td>22003</td>
<td>HY019</td>
<td></td>
</tr>
<tr>
<td>22008</td>
<td>22007</td>
<td></td>
</tr>
<tr>
<td>22005</td>
<td>22018</td>
<td></td>
</tr>
<tr>
<td>24000</td>
<td>07005</td>
<td></td>
</tr>
<tr>
<td>37000</td>
<td>42000</td>
<td></td>
</tr>
<tr>
<td>70100</td>
<td>HY018</td>
<td></td>
</tr>
<tr>
<td>S0001</td>
<td>42S01</td>
<td></td>
</tr>
<tr>
<td>S0002</td>
<td>42S02</td>
<td></td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY001</td>
<td>Returned for an invalid attribute value.</td>
<td></td>
</tr>
<tr>
<td>HY002</td>
<td>Returned for updating or deleting data by a call to SQLSetPos, or adding, updating, or deleting data by a call to SQLBulkOperations, when the concurrency is read-only.</td>
<td></td>
</tr>
<tr>
<td>HY003</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY004</td>
<td>Returned for an invalid attribute value.</td>
<td></td>
</tr>
<tr>
<td>HY008</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY009</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY010</td>
<td>SQLSTATE S1010 is mapped to SQLSTATE HY007 when SQLDescribeCol is called prior to calling SQLPrepare, SQLExecDirect, or a catalog function for the StatementHandle. Otherwise, SQLSTATE S1010 is mapped to SQLSTATE HY010.</td>
<td></td>
</tr>
<tr>
<td>HY011</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY012</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY090</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY091</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY092</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY096</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY097</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY098</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY099</td>
<td>Returned for an invalid use of a null pointer.</td>
<td></td>
</tr>
</tbody>
</table>

ODBC 2.x SQLSTATE S1002 is mapped to ODBC 3.x SQLSTATE 07009 if the underlying function is SQLBindCol, SQLColAttribute, SQLExtendedFetch, SQLFetch, SQLFetchScroll, or SQLGetData.

ODBC 3.x SQLSTATE 07009 is mapped to ODBC 2.x SQLSTATE S1093 if the underlying function is SQLBindParameter or SQLDescribeParam.
ODBC 3.x SQLSTATE 07008 is mapped to ODBC 2.x SQLSTATE S1000.

© 2016 Microsoft

Datetime Data Type Changes

In ODBC 3.x, the identifiers for date, time, and timestamp SQL data types have changed from SQL_DATE, SQL_TIME, and SQL_TIMESTAMP (with instances of `#define` in the header file of 9, 10, and 11) to SQL_TYPE_DATE, SQL_TYPE_TIME, and SQL_TYPE_TIMESTAMP (with instances of `#define` in the header file of 91, 92, and 93), respectively. The corresponding C type identifiers have changed from SQL_C_DATE, SQL_C_TIME, and SQL_C_TIMESTAMP to SQL_C_TYPE_DATE, SQL_C_TYPE_TIME, and SQL_C_TYPE_TIMESTAMP, respectively. The column size and decimal digits returned for the SQL datetime data types in ODBC 3.x are the same as the precision and scale returned for them in ODBC 2.x. These values are different than the values in the SQL_DESC_PRECISION and SQL_DESC_SCALE descriptor fields. (For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size.)

These changes affect `SQLDescribeCol`, `SQLDescribeParam`, and `SQLColAttribute`; `SQLBindCol`, `SQLBindParameter`, and `SQLGetData`; and `SQLColumns`, `SQLGetTypeInfo`, `SQLProcedureColumns`, `SQLStatistics`, and `SQLSpecialColumns`.

The following table shows how the ODBC 3.x Driver Manager performs mapping of the date, time, and timestamp C data types entered in the `TargetType` arguments of `SQLBindCol` and `SQLGetData` or in the `ValueType` argument of `SQLBindParameter`. 

<table>
<thead>
<tr>
<th>Target Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1100 HY100</td>
<td>2.x app</td>
</tr>
</tbody>
</table>
The following table shows how the ODBC 3.x Driver Manager performs mapping of the date, time, and timestamp SQL data types entered in the ParameterType argument of `SQLBindParameter` or in the DataType argument of `SQLGetTypeInfo`.

<table>
<thead>
<tr>
<th>Data type code entered</th>
<th>2.x app to 2.x driver</th>
<th>2.x app to 3.x driver</th>
<th>3.x app to 2.x driver</th>
<th>3.x app to 3.x driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_DATE (9)</td>
<td>No mapping</td>
<td>SQL_C_TYPE_DATE (91)</td>
<td>No mapping[1]</td>
<td>SQL_C_TYPE_DATE (91)</td>
</tr>
<tr>
<td>SQL_C_TYPE_DATE (91)</td>
<td>Error (from DM)</td>
<td>Error (from DM)</td>
<td>SQL_C_DATE (9)</td>
<td>No mapping[2]</td>
</tr>
<tr>
<td>SQL_C_TIME (10)</td>
<td>No mapping</td>
<td>SQL_C_TYPE_TIME (92)</td>
<td>No mapping[1]</td>
<td>SQL_C_TYPE_TIME (92)</td>
</tr>
<tr>
<td>SQL_C_TYPE_TIME (92)</td>
<td>Error (from DM)</td>
<td>Error (from DM)</td>
<td>SQL_C_TIME (10)</td>
<td>No mapping[2]</td>
</tr>
<tr>
<td>SQL_C_TIMESTAMP (11)</td>
<td>No mapping</td>
<td>SQL_C_TYPE_TIMESTAMP (93)</td>
<td>No mapping[1]</td>
<td>SQL_C_TYPE_TIMESTAMP (93)</td>
</tr>
<tr>
<td>SQL_C_TYPE_TIMESTAMP (93)</td>
<td>Error (from DM)</td>
<td>Error (from DM)</td>
<td>SQL_C_TIMESTAMP (11)</td>
<td>No mapping[2]</td>
</tr>
</tbody>
</table>

[1] As a result of this, an ODBC 3.x application working with an ODBC 2.x driver can use the date, time, or timestamp codes returned in the result sets that are returned by the catalog functions.

[2] As a result of this, an ODBC 3.x application working with an ODBC 3.x driver can use the date, time, or timestamp codes returned in the result sets that are returned by the catalog functions.
Writing ODBC 3.x Applications

When an ODBC 2.x application is upgraded to ODBC 3.x, it should be written such that it works with both ODBC 2.x and 3.x drivers. The application should incorporate conditional code to take full advantage of the ODBC 3.x features.

The SQL_ATTR_ODBC_VERSION environment attribute should be set to SQL_OV_ODBC2. This will ensure that the driver behaves like an ODBC 2.x driver with respect to the changes described in the section Behavioral Changes.

If the application will use any of the features described in the section New Features, conditional code should be used to determine whether the driver is an ODBC 3.x or ODBC 2.x driver. The application uses SQLGetDiagField and SQLGetDiagRec to obtain ODBC 3.x SQLSTATEs while doing error processing on these conditional code fragments.

The following points about the new functionality should be considered:

- An application affected by the change in rowset size behavior should be careful not to call SQLFetch when the array size is greater than 1. These applications should replace calls to SQLExtendedFetch with calls to SQLSetStmtAttr to set the SQL_ATTR_ARRAY_STATUS_PTR statement attribute and to SQLFetchScroll, so that they have common code that works with both ODBC 3.x and ODBC 2.x drivers. Because SQLSetStmtAttr with SQL_ATTR_ROW_ARRAY_SIZE will be mapped to SQLSetStmtAttr with SQL_ROWSET_SIZE for ODBC 2.x drivers, applications can just set SQL_ATTR_ROW_ARRAY_SIZE for their multirow fetch operations.

- Most applications that are upgrading are not actually affected by changes in SQLSTATE codes. For those applications that are affected, they can do a mechanical search and replace in most cases using the error conversion table in the "SQLSTATE Mapping" section to convert ODBC 3.x error codes to ODBC 2.x codes. Since the ODBC 3.x Driver Manager will perform mapping from ODBC 2.x SQLSTATEs to ODBC 3.x SQLSTATEs, these application writers need only check for the ODBC 3.x SQLSTATEs and not worry about including ODBC 2.x SQLSTATEs in conditional code.

- If an application makes great use of date, time, and timestamp data types, the application can declare itself to be an ODBC 2.x application and use its existing code instead of using conditioning code.

The upgrade should also include the following steps:

- Call SQLSetEnvAttr before allocating a connection to set the SQL_ATTR_ODBC_VERSION environment attribute to SQL_OV_ODBC2.

- Replace all calls to SQLAllocEnv, SQLAllocConnect, or SQLAllocStmt with calls to SQLAllocHandle with the appropriate HandleType argument of SQL_HANDLE_ENV, SQL_HANDLE_DBC, or SQL_HANDLE_STMT.

- Replace all calls to SQLFreeEnv or SQLFreeConnect with calls to SQLFreeHandle with the appropriate HandleType argument of SQL_HANDLE_DBC or SQL_HANDLE_STMT.

- Replace all calls to SQLSetConnectOption with calls to SQLSetConnectAttr. If setting an attribute whose value is a string, set the StringLength argument appropriately. Change Attribute argument from SQL_XXXX to SQL_ATTR_XXXX.

- Replace all calls to SQLGetConnectOption with calls to SQLGetConnectAttr. If getting a string or binary attribute, set BufferLength to the appropriate value and pass in a StringLength argument. Change Attribute argument from SQL_XXXX to SQL_ATTR_XXXX.

- Replace all calls to SQLSetStmtOption with calls to SQLSetStmtAttr. If setting an attribute whose value is a...
string, set the StringLength argument appropriately. Change Attribute argument from SQL_XXXX to SQL_ATTR_XXXX.

- Replace all calls to SQLGetStmtOption with calls to SQLGetStmtAttr. If getting a string or binary attribute, set BufferLength to the appropriate value and pass in a StringLength argument. Change Attribute argument from SQL_XXXX to SQL_ATTR_XXXX.

- Replace all calls to SQLTransact with calls to SQLEndTran. If the rightmost valid handle in the SQLTransact call is an environment handle, a HandleType argument of SQL_HANDLE_ENV should be used in the SQLEndTran call with the appropriate Handle argument. If the rightmost valid handle in your SQLTransact call is a connection handle, a HandleType argument of SQL_HANDLE_DBC should be used in the SQLEndTran call with the appropriate Handle argument.

- Replace all calls to SQLColAttributes with calls to SQLColAttribute. If the FieldIdentifier argument is either SQL_COLUMN_PRECISION, SQL_COLUMN_SCALE, or SQL_COLUMN_LENGTH, do not change anything other than the name of the function. If not, change FieldIdentifier from SQL_COLUMN_XXXX to SQL_DESC_XXXX. If FieldIdentifier is SQL_DESC_CONCISE_TYPE and the data type is a datetime data type, change to the corresponding ODBC 3.x data type.

- If using block cursors, scrollable cursors, or both, the application does the following:
  - Sets the rowset size, cursor type, and cursor concurrency using SQLSetStmtAttr.
  - Calls SQLSetStmtAttr to set SQL_ATTR_ROW_STATUS_PTR to point to an array of status records.
  - Calls SQLSetStmtAttr to set SQL_ATTR_ROWS_FETCHED_PTR to point to an SQLINTEGER.
  - Performs the required bindings and executes the SQL statement.
  - Calls SQLFetchScroll in a loop to fetch rows and move around in the result set.
  - If it wants to fetch by bookmark, the application calls SQLSetStmtAttr to set SQL_ATTR_FETCH_BOOKMARK_PTR to a variable that will contain the bookmark for the row that it wants to fetch, and calls SQLFetchScroll with a FetchOrientation argument of SQL_FETCH_BOOKMARK.

- If using arrays of parameters, the application does the following:
  - Calls SQLSetStmtAttr to set the SQL_ATTR_PARAMSET_SIZE attribute to the size of the parameter array.
  - Calls SQLSetStmtAttr to set SQL_ATTR_ROWS_PROCESSSED_PTR to point to an internal UDWORD variable.
  - Performs prepare, bind, and execute operations as appropriate.
  - If execution halts for some reason (such as SQL_NEED_DATA), it can find the "current" row of parameters by inspecting the location pointed to by SQL_ATTR_ROWS_PROCESSSED_PTR.

This section contains the following topics.

- Mapping Replacement Functions for Backward Compatibility of Applications
- Calling SQLCloseCursor
- Calling SQLGetDiagField
- Calling SQLSetPos
- Cursor Library Operations
- Mapping the Cursor Attributes1 Information Types
- SQL_NO_DATA
Mapping Replacement Functions for Backward Compatibility of Applications

An ODBC 3.x application working through the ODBC 3.x Driver Manager will work against an ODBC 2.x driver as long as no new features are used. Both duplicated functionality and behavioral changes do, however, affect the way that the ODBC 3.x application works on an ODBC 2.x driver. When working with an ODBC 2.x driver, the Driver Manager maps the following ODBC 3.x functions, which have replaced one or more ODBC 2.x functions, into the corresponding ODBC 2.x functions.

<table>
<thead>
<tr>
<th>ODBC 3.x function</th>
<th>ODBC 2.x function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocHandle</td>
<td>SQLAllocEnv, SQLAllocConnect, or SQLAllocStmt</td>
</tr>
<tr>
<td>SQLBulkOperations</td>
<td>SQLSetPos</td>
</tr>
<tr>
<td>SQLColAttribute</td>
<td>SQLColAttributes</td>
</tr>
<tr>
<td>SQLEndTran</td>
<td>SQLTransact</td>
</tr>
<tr>
<td>SQLFetch</td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td>SQLFetchScroll</td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td>SQLFreeHandle</td>
<td>SQLFreeEnv, SQLFreeConnect, or SQLFreeStmt</td>
</tr>
<tr>
<td>SQLGetConnectAttr</td>
<td>SQLGetConnectOption</td>
</tr>
<tr>
<td>SQLGetDiagRec</td>
<td>SQLError</td>
</tr>
<tr>
<td>SQLGetStmtAttr</td>
<td>SQLGetStmtOption[1]</td>
</tr>
<tr>
<td>SQLSetConnectAttr</td>
<td>SQLSetConnectOption</td>
</tr>
<tr>
<td>SQLSetStmtAttr</td>
<td>SQLSetStmtOption[1]</td>
</tr>
</tbody>
</table>

[1] Other actions might also be taken, depending on the attribute being requested.

**SQLAllocHandle**

The Driver Manager maps this to **SQLAllocEnv**, **SQLAllocConnect**, or **SQLAllocStmt**, as appropriate. The following call to **SQLAllocHandle**:

```c
SQLAllocHandle(HandleType, InputHandle, OutputHandlePtr);
```

will result in the Driver Manager performing the following (conceptual, no error checking) mapping:
SQLBulkOperations

The Driver Manager maps this to SQLSetPos. The following call to SQLBulkOperations:

```sql
SQLBulkOperations(hstmt, Operation);
```

will result in the following sequence of steps:

1. If the Operation argument is SQL_ADD, the Driver Manager calls SQLSetPos as follows:
   ```sql
   SQLSetPos (hstmt, 0, SQL_ADD, SQL_LOCK_NO_CHANGE);
   ```

2. If the Operation argument is not SQL_ADD, the driver returns SQLSTATE HY092 (Invalid attribute/option identifier).

3. If the application attempts to change the SQL_ATTR_ROW_STATUS_PTR between calls to SQLFetch or SQLFetchScroll and SQLBulkOperations, the Driver Manager will return SQLSTATE HY011 (Attribute cannot be set now).

4. If the Operation argument is SQL_ADD, the application must call SQLBindCol to bind the data to be inserted. It cannot call SQLSetDescField or SQLSetDescRec to bind the data to be inserted.

5. If the Operation argument is SQL_ADD and the number of rows to be inserted is not the same as the current rowset size, SQLSetStmtAttr must be called to set the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows to be inserted before calling SQLBulkOperations. To revert back to the previous rowset size, the application must set the SQL_ATTR_ROW_ARRAY_SIZE statement attribute before SQLFetch, SQLFetchScroll, or SQLSetPos is called.

SQLColAttribute

The Driver Manager maps this to SQLColAttributes. The following call to SQLColAttribute:

```sql
SQLColAttribute(StatementHandle, ColumnNumber, FieldIdentifier, CharacterAttributePtr, BufferLength,
```

will result in the following sequence of steps:

1. If FieldIdentifier is one of the following:
   ```sql
   SQL_DESC_PRECISION, SQL_DESC_SCALE, SQL_DESC_LENGTH, SQL_DESC_OCTET_LENGTH,
   SQL_DESC_UNNAMED, SQL_DESC_BASE_COLUMN_NAME, SQL_DESC_LITERAL_PREFIX,
   ```
the Driver Manager returns SQL_ERROR with SQLSTATE HY091 (Invalid descriptor field identifier). No further rules of this section apply.

2. The Driver Manager maps SQL_COLUMN_COUNT, SQL_COLUMN_NAME, or SQL_COLUMN_NULLABLE to SQL_DESC_COUNT, SQL_DESC_NAME, or SQL_DESC_NULLABLE, respectively. (An ODBC 2.x driver need only support SQL_COLUMN_COUNT, SQL_COLUMN_NAME, and SQL_COLUMN_NULLABLE, not SQL_DESC_COUNT, SQL_DESC_NAME, and SQL_DESC_NULLABLE.) The call to SQLColAttribute is mapped to:

```c
SQLColAttributes(StatementHandle, ColumnNumber, FieldIdentifier, CharacterAttributePtr, BufferLength);
```

3. All other FieldIdentifier values are passed through to the driver, with SQLColAttribute mapped to SQLColAttributes as shown previously.

4. If BufferLength is less than 0, the Driver Manager returns SQL_ERROR with SQLSTATE HY090 (Invalid string or buffer length). No further rules of this section apply.

5. If FieldIdentifier is SQL_DESC_CONCISE_TYPE and the returned type is a concise datetime data type, the Driver Manager maps the return values for date, time, and timestamp codes.

6. The Driver Manager performs necessary checks to see whether SQLSTATE HY010 (Function sequence error) needs to be raised. If so, the Driver Manager returns SQL_ERROR and SQLSTATE HY010 (Function sequence error). No further rules of this section apply.

### SQLEndTran

The Driver Manager maps this to SQLTransact. The following call to SQLEndTran:

```c
SQLEndTran(HandleType, Handle, CompletionType);
```

will result in the Driver Manager performing the following (conceptual, no error checking) mapping:

```c
switch (HandleType) {
    case SQL_HANDLE_ENV:return(SQLTransact(Handle, SQL_NULL_HDBC, CompletionType));
    case SQL_HANDLE_DBC:return(SQLTransact(SQL_NULL_HENV, Handle, CompletionType));
    default: // return SQL_ERROR, SQLSTATE HY092 ("Invalid attribute/option identifier")
}
```

### SQLFetch

The Driver Manager maps this to SQLExtendedFetch with a FetchOrientation argument of SQL_FETCH_NEXT. The following call to SQLFetch:

```c
SQLFetch (StatementHandle);
```
will result in the Driver Manager calling `SQLExtendedFetch`, as follows:

```
rc = SQLExtendedFetch(StatementHandle, FetchOrientation, FetchOffset, &RowCount, RowStatusArray);
```

In this call, the `pcRow` argument is set to the value that the application sets the `SQL_ATTR_ROWS_FETCHED_PTR` statement attribute to through a call to `SQLSetStmtAttr`.

**Note**

When the application calls `SQLSetStmtAttr` to set `SQL_ATTR_ROW_STATUS_PTR` to point to a status array, the Driver Manager caches the pointer. `RowStatusArray` can be equal to a null pointer.

If the driver does not support `SQLExtendedFetch` and the cursor library is loaded, the Driver Manager uses the cursor library's `SQLExtendedFetch` to map `SQLFetch` to `SQLExtendedFetch`. If the driver does not support `SQLExtendedFetch` and the cursor library is not loaded, the Driver Manager passes the call to `SQLFetch` through to the driver. If the application calls `SQLSetStmtAttr` to set `SQL_ATTR_ROW_STATUS_PTR`, the Driver Manager ensures that the array is populated. If the application calls `SQLSetStmtAttr` to set `SQL_ATTR_ROWS_FETCHED_PTR`, the Driver Manager sets this field to 1.

**SQLFetchScroll**

The Driver Manager maps this to `SQLExtendedFetch`. The following call to `SQLFetchScroll`:

```
SQLFetchScroll(StatementHandle, FetchOrientation, FetchOffset);
```

will result in the following sequence of steps:

1. When the application calls `SQLSetStmtAttr` to set `SQL_ATTR_ROW_STATUS_PTR` (which sets the `SQL_DESC_ARRAY_STATUS_PTR` field in the IRD) to point to a status array, the Driver Manager caches this pointer. Let this pointer be `RowStatusArray`; otherwise, let `RowStatusArray` be equal to a null pointer. If the `RowStatusArray` argument is set to a null pointer, the Driver Manager generates a row-status array.

2. If `FetchOrientation` is not one of `SQL_FETCH_NEXT`, `SQL_FETCH_PRIOR`, `SQL_FETCH_ABSOLUTE`, `SQL_FETCH_RELATIVE`, `SQL_FETCH_FIRST`, `SQL_FETCH_LAST`, or `SQL_FETCH_BOOKMARK`, the Driver Manager returns with SQL_ERROR and SQLSTATE HY106 (Fetch type out of range). No further rules of this section apply.

3. Case:

   - If `FetchOrientation` is equal to `SQL_FETCH_BOOKMARK`, then:
     
     - If `SQLSetStmtAttr` was called earlier to set the value of `SQL_ATTR_FETCH_BOOKMARK_PTR`, then let `Bmk` be the value obtained by dereferencing the pointer `SQL_DESC_FETCH_BOOKMARK_PTR`.
     
     - Otherwise, return SQL_ERROR with SQLSTATE HY111 (Invalid bookmark value). No further rules of this section apply.

The Driver Manager now calls `SQLExtendedFetch`, as follows:

```
rc = SQLExtendedFetch(StatementHandle, FetchOrientation, Bmk, pcRow, RowStatusArray);
```
Otherwise, the Driver Manager calls \texttt{SQLExtendedFetch}, as follows:

\begin{verbatim}
rc = SQLExtendedFetch(StatementHandle, FetchOrientation, FetchOffset, pcRow, RowStatusArray);
\end{verbatim}

In these calls, the \texttt{pcRow} argument is set to the value that the application sets the \texttt{SQL_ATTR_ROWS_FETCHED_PTR} statement attribute to through a call to \texttt{SQLSetStmtAttr}.

- \texttt{SQL_ATTR_ROW_ARRAY_SIZE} is mapped to \texttt{SQL_ROWSET_SIZE}.
- If \texttt{rc} is equal to \texttt{SQL_SUCCESS} or \texttt{SQL_SUCCESS_WITH_INFO}, and if \texttt{FetchOrientation} is equal to \texttt{SQL_FETCH_BOOKMARK} and \texttt{FetchOffset} is not equal to 0, then the Driver Manager posts a warning, \texttt{SQLSTATE 01S10 (Attempt to fetch by a bookmark offset, offset value ignored)}, and returns \texttt{SQL_SUCCESS_WITH_INFO}.

**SQLFreeHandle**

The Driver Manager maps this to \texttt{SQLFreeEnv}, \texttt{SQLFreeConnect}, or \texttt{SQLFreeStmt} as appropriate. The following call to \texttt{SQLFreeHandle}:

\begin{verbatim}
SQLFreeHandle(HandleType, Handle);
\end{verbatim}

will result in the Driver Manager performing the following (conceptual, no error checking) mapping:

\begin{verbatim}
switch (HandleType) {
    case SQL_HANDLE_ENV: return (SQLFreeEnv(Handle));
    case SQL_HANDLE_DBC: return (SQLFreeConnect(Handle));
    case SQL_HANDLE_STMT: return (SQLFreeStmt(Handle, SQL_DROP));
    default: // return SQL_ERROR, SQLSTATE HY092 ("Invalid attribute/option identifier")
}
\end{verbatim}

**SQLGetConnectAttr**

The Driver Manager maps this to \texttt{SQLGetConnectOption}. The following call to \texttt{SQLGetConnectAttr}:

\begin{verbatim}
SQLGetConnectAttr(ConnectionHandle, Attribute, ValuePtr, BufferLength, StringLengthPtr);
\end{verbatim}

will result in the following sequence of steps:

1. If \texttt{Attribute} is not a driver-defined connection or statement attribute and is not an attribute defined in ODBC 2.\textit{x}, the Driver Manager returns \texttt{SQL_ERROR} with \texttt{SQLSTATE HY092 (Invalid attribute/option identifier)}. No further rules in this section apply.

2. If \texttt{Attribute} is equal to \texttt{SQL_ATTR_AUTO_IPD} or \texttt{SQL_ATTR_METADATA_ID}, the Driver Manager returns \texttt{SQL_ERROR} with \texttt{SQLSTATE HY092 (Invalid attribute/option identifier)}.

3. The Driver Manager performs necessary checks to see if \texttt{SQLSTATE 08003 (Connection not open) or}
SQLSTATE HY010 (Function sequence error) needs to be raised. If so, the Driver Manager returns SQL_ERROR and posts the appropriate error message. No further rules of this section apply.

4. The Driver Manager calls `SQLGetConnectOption` as follows:

```
SQLGetConnectOption (ConnectionHandle, Attribute, ValuePtr);
```

Note that the `BufferLength` and `StringLengthPtr` are ignored.

**SQLGetData**

When an ODBC 3.x application working with an ODBC 2.x driver calls `SQLGetData` with the `ColumnNumber` argument equal to 0, the ODBC 3.x Driver Manager maps this to a call to `SQLGetStmtOption` with the `Option` attribute set to SQL_GET_BOOKMARK.

**SQLGetStmtAttr**

The Driver Manager maps this to `SQLGetStmtOption`. The following call to `SQLGetStmtAttr`:

```
SQLGetStmtAttr(StatementHandle, Attribute, ValuePtr, BufferLength, StringLengthPtr);
```

will result in the following sequence of steps:

1. If `Attribute` is not a driver-defined connection or statement attribute and is not an attribute defined in ODBC 2.x, the Driver Manager returns SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier). No further rules in this section apply.

2. If `Attribute` is one of the following:
   - `SQL_ATTR_APP_ROW_DESC`
   - `SQL_ATTR_APP_PARAM_DESC`
   - `SQL_ATTR_AUTO_IPD`
   - `SQL_ATTR_ROW_BIND_TYPE`
   - `SQL_ATTR_IMP_ROW_DESC`
   - `SQL_ATTR_IMP_PARAM_DESC`
   - `SQL_ATTR_METADATA_ID`
   - `SQL_ATTR_PARAM_BIND_TYPE`
   - `SQL_ATTR_PREDICATE_PTR`
   - `SQL_ATTR_PREDICATE_OCTET_LENGTH_PTR`
   - `SQL_ATTR_PARAM_BIND_OFFSET_PTR`
   - `SQL_ATTR_ROW_BIND_OFFSET_PTR`
   - `SQL_ATTR_APP_ROW_DESC`
   - `SQL_ATTR_APP_PARAM_DESC`
   - `SQL_ATTR_AUTO_IPD`
   - `SQL_ATTR_ROW_BIND_TYPE`
   - `SQL_ATTR_IMP_ROW_DESC`
   - `SQL_ATTR_IMP_PARAM_DESC`
   - `SQL_ATTR_METADATA_ID`
   - `SQL_ATTR_PARAM_BIND_TYPE`
   - `SQL_ATTR_PREDICATE_PTR`
   - `SQL_ATTR_PREDICATE_OCTET_LENGTH_PTR`
   - `SQL_ATTR_PARAM_BIND_OFFSET_PTR`
   - `SQL_ATTR_ROW_BIND_OFFSET_PTR`
The Driver Manager returns SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier). No further rules of this section apply.

3. The Driver Manager performs necessary checks to see whether SQLSTATE HY010 (Function sequence error) needs to be raised. If so, the Driver Manager returns SQL_ERROR and SQLSTATE HY010 (Function sequence error). No further rules of this section apply.

4. If Attribute is equal to SQL_ATTR_ROW_OPERATION_PTR, the Driver Manager returns SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier). No further rules of this section apply.

5. If Attribute is equal to SQL_ATTR_PARAM_OPERATION_PTR, the Driver Manager returns SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier). No further rules of this section apply.

6. If Attribute is equal to SQL_ATTR_ROWS_FETCHED_PTR, the Driver Manager returns a pointer to the internal Driver Manager variable cRow, which it has used or will use in a call to SQLExtendedFetch. No further rules of this section apply.

7. If Attribute is equal to SQL_ATTR_DESC_FETCH_BOOKMARK_PTR, the Driver Manager returns the appropriate pointer that it had cached during a call to SQLGetStmtAttr.

6. If Attribute is equal to SQL_ATTR_ROW_STATUS_PTR, the Driver Manager returns the appropriate pointer that it had cached during a call to SQLSetStmtAttr.

7. The Driver Manager calls SQLGetStmtOption as follows:

```sql
SQLGetStmtOption (hstmt, fOption, pvParam);
```

where hstmt, fOption, and pvParam will be set to the values of StatementHandle, Attribute, and ValuePtr, respectively. The BufferLength and StringLengthPtr are ignored.

## SQLSetConnectAttr

The Driver Manager maps this to SQLSetConnectOption. The following call to SQLSetConnectAttr:

```sql
SQLSetConnectAttr(ConnectionHandle, Attribute, ValuePtr, StringLength);
```

will result in the following sequence of steps:

1. If Attribute is not a driver-defined connection or statement attribute and is not an attribute defined in ODBC 2.x, the Driver Manager returns SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier). No further rules in this section apply.

2. If Attribute is equal to SQL_ATTR_AUTO_IPD, the Driver Manager returns SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier).

3. The Driver Manager performs necessary checks to see whether SQLSTATE 08003 (Connection not open) or SQLSTATE HY010 (Function sequence error) need to be raised. If one of these errors needs to be raised, the Driver Manager returns SQL_ERROR and posts the appropriate error message. No further rules of this section apply.

4. The Driver Manager calls SQLSetConnectOption as follows:

```sql
SQLSetConnectOption (hdbc, fOption, vParam);
```
where hdbc, fOption, and vParam will be set to the values of ConnectionHandle, Attribute, and ValuePtr, respectively. StringLengthPtr is ignored.

**Note**

The ability to set statement attributes on the connection level has been deprecated. Statement attributes should never be set on the connection level by an ODBC 3.x application.

### SQLSetStmtAttr

The Driver Manager maps this to SQLSetStmtOption. The following call to SQLSetStmtAttr:

```sql
SQLSetStmtAttr(StatementHandle, Attribute, ValuePtr, StringLength);
```

will result in the following sequence of steps:

1. If Attribute is not a driver-defined connection or statement attribute and is not an attribute defined in ODBC 2.x, the Driver Manager returns SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier). No further rules in this section apply.

2. If Attribute is one of the following:
   
   - SQL_ATTR_APP_ROW_DESC
   - SQL_ATTR_APP_PARAM_DESC
   - SQL_ATTR_AUTO_IPD
   - SQL_ATTR_ROW_BIND_TYPE
   - SQL_ATTR_IMP_ROW_DESC
   - SQL_ATTR_IMP_PARAM_DESC
   - SQL_ATTR_METADATA_ID
   - SQL_ATTR_PARAM_BIND_TYPE
   - SQL_ATTR_PREDICATE_PTR
   - SQL_ATTR_PREDICATE_OCTET_LENGTH_PTR
   - SQL_ATTR_PARAM_BIND_OFFSET_PTR
   - SQL_ATTR_ROW_BIND_OFFSET_PTR
   - SQL_ATTR_ROW_OPERATION_PTR
   - SQL_ATTR_PARAM_OPERATION_PTR

   the Driver Manager returns SQL_ERROR with SQLSTATE HY092 (Invalid attribute/option identifier). No further rules of this section apply.

3. The Driver Manager performs the necessary checks to see whether SQLSTATE HY010 (Function sequence error) need to be raised. If so, the Driver Manager returns SQL_ERROR and SQLSTATE HY010 (Function sequence error). No further rules of this section apply.

4. If Attribute is equal to SQL_ATTR_PARAMSET_SIZE or SQL_ATTR_PARAMS_PROCESSED_PTR, see the
section "Mappings for Handling Parameter Arrays," later in this topic. No further rules of this section apply.

5. If Attribute is equal to SQL_ATTR_ROWS_FETCHED_PTR, the Driver Manager caches the pointer value for later use with SQLFetchScroll.

6. If Attribute is equal to SQL_ATTR_ROW_STATUS_PTR, the Driver Manager caches the pointer value for later use with SQLFetchScroll or SQLSetPos. No further rules of this section apply.

7. If Attribute is equal to SQL_ATTR_FETCH_BOOKMARK_PTR, the Driver Manager caches ValuePtr and will use the cached value later in a call to SQLFetchScroll. No further rules of this section apply.

8. The Driver Manager calls SQLSetStmtOption as follows:

   ```
   SQLSetStmtOption (hstmt, fOption, vParam);
   ```

   where hstmt, fOption, and vParam will be set to the values of StatementHandle, Attribute, and ValuePtr, respectively. The StringLength argument is ignored.

   If an ODBC 2.x driver supports character-string, driver-specific statement options, an ODBC 3.x application should call SQLSetStmtOption to set those options.

Mappings for Handling Parameter Arrays

When the application calls:

```
SQLSetStmtAttr (StatementHandle, SQL_ATTR_PARAMSET_SIZE, Size, StringLength);
```

the Driver Manager calls:

```
SQLParamOptions (StatementHandle, Size, &RowCount);
```

The Driver Manager later returns a pointer to this variable when the application calls SQLGetStmtAttr to retrieve SQL_ATTR_PARAMS_PROCESSED_PTR. The Driver Manager cannot change this internal variable until the statement handle is returned to the prepared or allocated state.

An ODBC 3.x application can call SQLSetStmtAttr to obtain the value of SQL_ATTR_PARAMS_PROCESSED_PTR even though it has not explicitly set the SQL_DESC_ARRAY_SIZE field in the APD. This situation could arise, for example, if the application has a generic routine that checks for the current "row" of parameters being processed when SQLExecute returns SQL_NEED_DATA. This routine is invoked whether or not the SQL_DESC_ARRAY_SIZE is 1 or is greater than 1. To account for this, the Driver Manager will need to define this internal variable whether or not the application has called SQLSetStmtAttr to set the SQL_DESC_ARRAY_SIZE field in APD. If SQL_DESC_ARRAY_SIZE has not been set, the Driver Manager has to make sure that this variable contains the value 1 prior to returning from SQLExecDirect or SQLExecute.

Error Handling

In ODBC 3.x, calling SQLFetch or SQLFetchScroll populates the SQL_DESC_ARRAY_STATUS_PTR in the IRD, and the SQL_DIAG_ROW_NUMBER field of a given diagnostic record contains the number of the row in the rowset that this record pertains to. Using this, the application can correlate an error message with a given row position.
An ODBC 2.x driver will be unable to provide this functionality. However, it will provide error demarcation with SQLSTATE 01S01 (Error in row). An ODBC 3.x application that is using SQLFetch or SQLFetchScroll while going against an ODBC 2.x driver needs to be aware of this fact. Note also that such an application will be unable to call SQLGetDiagField to actually get the SQL_DIAG_ROW_NUMBER field anyway. An ODBC 3.x application working with an ODBC 2.x driver will be able to call SQLGetDiagField only with a DiagIdentifier argument of SQL_DIAG_MESSAGE_TEXT, SQL_DIAG_NATIVE, SQL_DIAG_RETURNCODE, or SQL_DIAG_SQLSTATE. The ODBC 3.x Driver Manager maintains the diagnostic data structure when working with an ODBC 2.x driver, but the ODBC 2.x driver returns only these four fields.

When an ODBC 2.x application is working with an ODBC 2.x driver, if an operation can cause multiple errors to be returned by the Driver Manager, different errors may be returned by the ODBC 3.x Driver Manager than by the ODBC 2.x Driver Manager.

Mappings for Bookmark Operations

The ODBC 3.x Driver Manager performs the following mappings when an ODBC 3.x application working with an ODBC 2.x driver performs bookmark operations.

**SQLBindCol**

When an ODBC 3.x application working with an ODBC 2.x driver calls SQLBindCol to bind to column 0 with fCType equal to SQL_C_VARBOOKMARK, the ODBC 3.x Driver Manager checks to see whether the BufferLength argument is less than 4 or greater than 4, and if so, returns SQLSTATE HY090 (Invalid string or buffer length). If the BufferLength argument is equal to 4, the Driver Manager calls SQLBindCol in the driver, after replacing fCType with SQL_C_BOOKMARK.

**SQLColAttribute**

When an ODBC 3.x application working with an ODBC 2.x driver calls SQLColAttribute with the ColumnNumber argument set to 0, the Driver Manager returns the FieldIdentifier values listed in the following table.

<table>
<thead>
<tr>
<th>FieldIdentifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_AUTO_UNIQUE_VALUE</td>
<td>SQL_FALSE</td>
</tr>
<tr>
<td>SQL_DESC_CASE_SENSITIVE</td>
<td>SQL_FALSE</td>
</tr>
<tr>
<td>SQL_DESC_CATALOG_NAME</td>
<td>&quot;&quot; (empty string)</td>
</tr>
<tr>
<td>SQL_DESC_CONCISE_TYPE</td>
<td>SQL_BINARY</td>
</tr>
<tr>
<td>SQL_DESC_COUNT</td>
<td>The same value returned by SQLNumResultCols</td>
</tr>
<tr>
<td>SQL_DESC_DATETIME_INTERVAL_CODE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DESC_DISPLAY_SIZE</td>
<td>8</td>
</tr>
<tr>
<td>SQL_DESC_FIXED_PREC_SCALE</td>
<td>SQL_FALSE</td>
</tr>
<tr>
<td>SQL_DESC_LABEL</td>
<td>&quot;&quot; (empty string)</td>
</tr>
<tr>
<td>SQL_DESC_LENGTH</td>
<td>0</td>
</tr>
</tbody>
</table>
When an ODBC 3.x application working with an ODBC 2.x driver calls `SQLDescribeCol` with the `ColumnNumber` argument set to 0, the Driver Manager returns the values listed in the following table.

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColumnName</td>
<td>&quot;&quot; (empty string)</td>
</tr>
<tr>
<td>*NameLengthPtr</td>
<td>0</td>
</tr>
<tr>
<td>*DataTypePtr</td>
<td>SQL_BINARY</td>
</tr>
<tr>
<td>*ColumnSizePtr</td>
<td>4</td>
</tr>
<tr>
<td>*DecimalDigitsPtr</td>
<td>0</td>
</tr>
<tr>
<td>*NullablePtr</td>
<td>SQL_NO_NULLS</td>
</tr>
</tbody>
</table>

**SQLDescribeCol**

When an ODBC 3.x application working with an ODBC 2.x driver calls `SQLDescribeCol` with the `ColumnNumber` argument set to 0, the Driver Manager returns the values listed in the following table.
**SQLGetData**

When an ODBC 3.x application working with an ODBC 2.x driver makes the following call to `SQLGetData` to retrieve a bookmark:

```c
SQLGetData(StatementHandle, 0, SQL_C_VARBOOKMARK, TargetValuePtr, BufferLength, StrLen_or_IndPtr);
```

the call is mapped to `SQLGetStmtOption` with an `fOption` of SQL_GET_BOOKMARK, as follows:

```c
SQLGetStmtOption(hstmt, SQL_GET_BOOKMARK, TargetValuePtr)
```

where `hstmt` and `pvParam` are set to the values in `StatementHandle` and `TargetValuePtr`, respectively. The bookmark is returned in the buffer pointed to by the `pvParam`(`TargetValuePtr`) argument. The value in the buffer pointed to by the `StrLen_or_IndPtr` argument in the call to `SQLGetData` is set to 4.

This mapping is necessary to account for the case in which `SQLFetch` was called prior to the call to `SQLGetData` and the ODBC 2.x driver did not support `SQLExtendedFetch`. In this case, `SQLFetch` would be passed through to the ODBC 2.x driver, in which case bookmark retrieval is not supported.

`SQLGetData` cannot be called multiple times in an ODBC 2.x driver to retrieve a bookmark in parts, so calling `SQLGetData` with the `BufferLength` argument set to a value less than 4 and the `ColumnNumber` argument set to 0 will return SQLSTATE HY090 (Invalid string or buffer length). `SQLGetData` can, however, be called multiple times to retrieve the same bookmark.

**SQLSetStmtAttr**

When an ODBC 3.x application working with an ODBC 2.x driver calls `SQLSetStmtAttr` to set the SQL_ATTR_USE_BOOKMARKS attribute to SQL_UB_VARIABLE, the Driver Manager sets the attribute to SQL_UB_ON in the underlying ODBC 2.x driver.

**Calling SQLCloseCursor**

Because `SQLCloseCursor` is almost the same as `SQLFreeStmt` with SQL_CLOSE, the Driver Manager does not map this function. Replacement functions are mapped so that existing ODBC 2.x applications can easily move to ODBC 3.x by using the new functions. Such a move makes it easier for such applications to begin using new ODBC 3.x functionality inside of conditional code in a modular fashion. `SQLCloseCursor` does not represent any new functionality. An application does not gain any advantage by moving to `SQLCloseCursor` from `SQLFreeStmt` with SQL_CLOSE.
Calling SQLGetDiagField

When an ODBC 3.x application calls `SQLGetDiagField` in an ODBC 2.x driver, the driver will return SQL_SUCCESS and the appropriate information in `*DiagInfoPtr` if the `DiagIdentifier` argument is SQL_DIAG_CLASS_ORIGIN, SQL_DIAG_CLASS_SUBCLASS_ORIGIN, SQL_DIAG_CONNECTION_NAME, SQL_DIAG_MESSAGE_TEXT, SQL_DIAG_NATIVE, SQL_DIAG_NUMBER, SQL_DIAG_RETURNCODE, SQL_DIAG_SERVER_NAME, or SQL_DIAG_SQLSTATE. All other diagnostic fields will return SQL_ERROR.

Calling SQLSetPos

In ODBC 2.x, the pointer to the row status array was an argument to `SQLExtendedFetch`. The row status array was later updated by a call to `SQLSetPos`. Some drivers have relied on the fact that this array does not change between `SQLExtendedFetch` and `SQLSetPos`. In ODBC 3.x, the pointer to the status array is a descriptor field and therefore the application can easily change it to point to a different array. This can be a problem when an ODBC 3.x application is working with an ODBC 2.x driver but is calling `SQLSetStmtAttr` to set the array status pointer and is calling `SQLFetchScroll` to fetch data. The Driver Manager maps it as a sequence of calls to `SQLExtendedFetch`. In the following code, an error would normally be raised when the Driver Manager maps the second `SQLSetStmtAttr` call when working with an ODBC 2.x driver:

```sql
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, rgfRowStatus, 0);
SQLFetchScroll(hstmt, fFetchType, iRow);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, rgfRowStatus1, 0);
SQLSetPos(hstmt, iRow, fOption, fLock);
```

The error would be raised if there were no way to change the row status pointer in ODBC 2.x between calls to `SQLExtendedFetch`. Instead, the Driver Manager performs the following steps when working with an ODBC 2.x driver:

1. Initializes an internal Driver Manager flag `fSetPosError` to TRUE.
2. When an application calls `SQLFetchScroll`, the Driver Manager sets `fSetPosError` to FALSE.
3. When the application calls `SQLSetStmtAttr` to set `SQL_ATTR_ROW_STATUS_PTR`, the Driver Manager sets `fSetPosError` equal to TRUE.
4. When the application calls `SQLSetPos`, with `fSetPosError` equal to TRUE, the Driver Manager raises SQL_ERROR with SQLSTATE HY011 (Attribute cannot be set now) to indicate that the application attempted to call `SQLSetPos` after changing the row status pointer but prior to calling `SQLFetchScroll`.

Cursor Library Operations
Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

If an application working with an ODBC 2.x driver makes calls to the ODBC 3.x cursor library, the application might be able to use ODBC 3.x features that are not supported by the ODBC 2.x driver. An application writer should be careful how these features are used, however. Use of the ODBC 3.x cursor library does not make an ODBC 2.x driver into an ODBC 3.x driver.

© 2016 Microsoft

Mapping the Cursor Attributes 1

Information Types

When an ODBC 3.x application calls SQLGetInfo in an ODBC 2.x driver with the SQL_XXXX_CURSOR_ATTRIBUTES1 information type (for dynamic, forward-only, keyset-driver, or static cursors), the setting of the bits returned by Driver Manager depends on what the ODBC 2.x driver returns for the corresponding ODBC 2.x information types. The bits are set as shown in the following table.

<table>
<thead>
<tr>
<th>Bit in SQL_XXXX_CURSOR_ATTRIBUTES1</th>
<th>Cursor type</th>
<th>ODBC 2.x information type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CA1_NEXT</td>
<td>All</td>
<td>SQL_FETCH_DIRECTION</td>
</tr>
<tr>
<td>SQL_CA1_ABSOLUTE SQL_CA1_RELATIVE SQL_CA1_BOOKMARK</td>
<td>Dynamic, keyset-driver, static</td>
<td>SQL_FETCH_DIRECTION</td>
</tr>
<tr>
<td>SQL_CA1_LOCK_NO_CHANGE SQL_CA1_LOCK_UNLOCK SQL_CA1_LOCK_EXCLUSIVE</td>
<td>Dynamic, keyset-driver, static</td>
<td>SQL_LOCK_TYPES</td>
</tr>
<tr>
<td>SQL_CA1_POSITIONED_UPDATE SQL_CA1_POSITIONED_DELETE SQL_CA1_SELECT_FOR_UPDATE</td>
<td>All</td>
<td>SQL_POSITIONED_STATEMENTS</td>
</tr>
<tr>
<td>SQL_CA1_POS_POSITION SQL_CA1_POS_DELETE SQL_CA1_POS_REFRESH SQL_CA1_POS_BULK_ADD</td>
<td>Dynamic, keyset-driver, static</td>
<td>SQL_POS_OPERATIONS</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

SQL_NO_DATA

When an ODBC 3.x application calls SQLExecDirect, SQLExecute, or SQLParamData in an ODBC 2.x driver to
execute a searched update or delete statement that does not affect any rows at the data source, the driver should return SQL_SUCCESS, not SQL_NO_DATA. When an ODBC 2.x or ODBC 3.x application working with an ODBC 3.x driver calls SQLExecDirect, SQLExec, or SQLParamData with the same result, the ODBC 3.x driver should return SQL_NO_DATA.

© 2016 Microsoft

## Writing ODBC 3.x Drivers

The following table shows function support in an ODBC 3.x driver and an ODBC application, and the mapping performed by the Driver Manager when the functions are called against an ODBC 3.x driver.

<table>
<thead>
<tr>
<th>Function</th>
<th>Supported by an ODBC 3.x driver?</th>
<th>Supported by an ODBC 3.x application?</th>
<th>Mapped/supported by the ODBC 3.x Driver Manager to an ODBC 3.x driver?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocConnect</td>
<td>No</td>
<td>No[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLAllocEnv</td>
<td>No</td>
<td>No[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLAllocHandle</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLAllocStmt</td>
<td>No</td>
<td>No[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLBindCol</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLBindParam</td>
<td>No</td>
<td>Yes[2]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLBindParameter</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLBrowseConnect</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLBulkOperations</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLCancel</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLCloseCursor</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLColAttribute</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLColAttributes</td>
<td>No[3]</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLColumnPrivileges</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLColumns</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Function</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>SQLConnect</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLCopyDesc</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes[4]</td>
</tr>
<tr>
<td>SQLDataSources</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLDescribeCol</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLDescribeParam</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLDisconnect</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLDriverConnect</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLDrivers</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLEndTran</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLError</td>
<td>No</td>
<td>No[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLExecDirect</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLExecute</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLExtendedFetch</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SQLFetch</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLFetchScroll</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLForeignKeys</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLFreeConnect</td>
<td>No</td>
<td>Yes[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLFreeEnv</td>
<td>No</td>
<td>Yes[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLFreeHandle</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLFreeStmt</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLGetConnectAttr</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLGetCursorName</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLGetData</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLGetDescField</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLGetDescRec</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLGetDiagField</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLGetDiagRec</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Function</td>
<td>Yes</td>
<td>No[6]</td>
<td>Yes</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>SQLGetEnvAttr</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLGetFunctions</td>
<td>No[6]</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLGetInfo</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLGetStmtAttr</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLGetStmtOption</td>
<td>No[5]</td>
<td>No[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLGetTypeInfo</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLMoreResults</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLNativeSql</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLNumParams</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLNumResultCols</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLParamData</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLParamOptions</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLPrepare</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLPrimaryKeys</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLProcedureColumns</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLProcedures</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLPutData</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLRowCount</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetConnectAttr</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetConnectOption</td>
<td>No[5]</td>
<td>No[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetCursorName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetDescField</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetDescRec</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetEnvAttr</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetPos</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetParam</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetScrollOption</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetStmtAttr</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSetStmtOption</td>
<td>No[5]</td>
<td>No[1]</td>
<td>Yes</td>
</tr>
<tr>
<td>SQLSpecialColumns</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----</td>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td>SQLStatistics</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLTablePrivileges</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLTables</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQLTransact</td>
<td>No</td>
<td>No[1]</td>
<td>Yes</td>
</tr>
</tbody>
</table>

[1] This function is deprecated in ODBC 3.x. ODBC 3.x applications should not use this function. However, an Open Group or ISO CLI–compliant application can call this function.

[2] ODBC 3.x applications should use SQLBindParameter instead of SQLBindParam. However, an Open Group or ISO CLI–compliant application can call this function.

[3] Driver writers should note that the ODBC 2.x column attributes SQL_COLUMN_PRECISION, SQL_COLUMN_SCALE, and SQL_COLUMN_LENGTH must be supported with SQLO ColAttribute.

[4] SQLCopyDesc is partially implemented by the Driver Manager when a descriptor is being copied across connections that belong to different drivers. Drivers are required to support SQLCopyDesc across two of their own connections. Functions such as SQLDrivers, which are implemented solely by the Driver Manager, do not show up on this list.

[5] Under certain circumstances, drivers may need to support this function. For more information, see this function's reference page.

[6] The driver can choose to support SQLGetFunctions if the set of functions that the driver supports varies from connection to connection.

© 2016 Microsoft

**ODBC in Windows**

The following items apply only to ODBC running in Microsoft® Windows NT®/Windows 2000 and Microsoft Windows® 95/98 operating systems.

This section contains the following topics.

- Standards-Compliant Applications and Drivers
- Header Files
- Passing CString Objects to ODBC Functions
- Creating and Terminating Threads

© 2016 Microsoft

**Standards-Compliant Applications and Drivers**
A standards-compliant application or driver is one that conforms to the Open Group CAE Specification "Data Management: SQL Call-Level Interface (CLI)," and the ISO/IEC 9075-3:1995 (E) Call-Level Interface (SQL/CLI).

ODBC 3.x guarantees the following features:

- An application written to the Open Group and ISO CLI specifications will work with an ODBC 3.x driver or a standards-compliant driver when it is compiled with the ODBC 3.x header files and linked with ODBC 3.x libraries, and when it gains access to the driver through the ODBC 3.x Driver Manager.

- A driver written to the Open Group and ISO CLI specifications will work with an ODBC 3.x application or a standards-compliant application when it is compiled with the ODBC 3.x header files and linked with ODBC 3.x libraries, and when the application gains access to the driver through the ODBC 3.x Driver Manager.

Standards-compliant applications and drivers are compiled with the ODBC_STD compile flag.

Standards-compliant applications exhibit the following behavior:

- If a standards-compliant application calls SQLAllocEnv (which can occur because SQLAllocEnv is a valid function in the Open Group and ISO CLI), the call is mapped to SQLAllocHandleStd at compile time. As a result, at run time, the application calls SQLAllocHandleStd. During the course of processing this call, the Driver Manager sets the SQL_ATTR_ODBC_VERSION environment attribute to SQL_OV_ODBC3. A call to SQLAllocHandleStd is equivalent to a call to SQLAllocHandle with a HandleType of SQL_HANDLE_ENV and a call to SQLSetEnvAttr to set SQL_ATTR_ODBC_VERSION to SQL_OV_ODBC3.

- If a standards-compliant application calls SQLBindParam (which can occur because SQLBindParam is a valid function in the Open Group and ISO CLI), the ODBC 3.x Driver Manager maps the call to the equivalent call in SQLBindParameter. (See SQLBindParam Mapping in Appendix G: Driver Guidelines for Backward Compatibility.)

- To align with the ISO CLI, the ODBC 3.x header files contain aliases for information types used in calls to SQLGetInfo. A standards-compliant application can use these aliases instead of the ODBC 3.x information types. For more information, see the next topic, Header Files.

- A standards-compliant application must verify that all features it supports are supported in the driver it will work with. Setting the SQL_ATTR_CURSOR_SCROLLABLE statement attribute to SQL_SCROLLABLE and setting the SQL_ATTR_CURSOR_SENSITIVITY statement attribute to SQL_INSENSITIVE or SQL_SENSITIVE are capabilities that are available as optional features in the standards but are not included in the ODBC 3.x Core level and therefore might not be supported by all ODBC 3.x drivers. If a standards-compliant application uses these capabilities, it should verify that the driver that it will work with supports them.

© 2016 Microsoft

Header Files

The Sql.h header file contains prototypes for the functions and features in the Core ODBC Interface conformance level. The Sqlext.h header file contains prototypes for the functions and features in the Level 1 and Level 2 API conformance levels. The Sqltypes.h header file contains type definitions and indicators for the SQL data types.

The header files all contain a #define, ODBCVER, that an application or driver can set to be compiled for different versions of ODBC.

To align with the ISO CLI and Open Group CLI, the header files contain aliases for the information types used in calls to SQLGetInfo. In the following table, the column "ODBC name" indicates the ODBC name for the information type in ODBC API Reference. The column "Alias in header file" indicates the name that is used in the ISO CLI and the Open Group CLI. The actual numeric value of these manifest names is the same in both ODBC and the standard CLIs. These aliases enable a standards-compliant application or driver to compile with the ODBC 3.x header files.
These aliases include expansions of abbreviations in the ODBC names so that the names are more understandable. "MAX" is expanded to "MAXIMUM", "LEN" to "LENGTH", "MULT" to "MULTIPLE", "OJ" to "OUTER_JOIN", and "TXN" to "TRANSACTION."

<table>
<thead>
<tr>
<th>ODBC name</th>
<th>Alias in header file</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_MAX_CATALOG_NAME_LEN</td>
<td>SQL_MAXIMUM_CATALOG_NAME_LENGTH</td>
</tr>
<tr>
<td>SQL_MAX_COLUMN_NAME_LEN</td>
<td>SQL_MAXIMUM_COLUMN_NAME_LENGTH</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_GROUP_BY</td>
<td>SQL_MAXIMUM_COLUMNS_IN_GROUP_BY</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_ORDER_BY</td>
<td>SQL_MAXIMUM_COLUMNS_IN_ORDER_BY</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_SELECT</td>
<td>SQL_MAXIMUM_COLUMNS_IN_SELECT</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_TABLE</td>
<td>SQL_MAXIMUM_COLUMNS_IN_TABLE</td>
</tr>
<tr>
<td>SQL_MAX_CONCURRENT_ACTIVITIES</td>
<td>SQL_MAXIMUM_CONCURRENT_ACTIVITIES</td>
</tr>
<tr>
<td>SQL_MAX_CURSOR_NAME_LEN</td>
<td>SQL_MAXIMUM_CURSOR_NAME_LENGTH</td>
</tr>
<tr>
<td>SQL_MAX_DRIVER_CONNECTIONS</td>
<td>SQL_MAXIMUM_DRIVER_CONNECTIONS</td>
</tr>
<tr>
<td>SQL_MAX_IDENTIFIER_LEN</td>
<td>SQL_MAXIMUM_IDENTIFIER_LENGTH</td>
</tr>
<tr>
<td>SQL_MAX_SCHEMA_NAME_LEN</td>
<td>SQL_MAXIMUM_SCHEMA_NAME_LENGTH</td>
</tr>
<tr>
<td>SQL_MAX_STATEMENT_LEN</td>
<td>SQL_MAXIMUM_STATEMENT_LENGTH</td>
</tr>
<tr>
<td>SQL_MAX_TABLE_NAME_LEN</td>
<td>SQL_MAXIMUM_TABLE_NAME_LENGTH</td>
</tr>
<tr>
<td>SQL_MAX_TABLES_IN_SELECT</td>
<td>SQL_MAXIMUM_TABLES_IN_SELECT</td>
</tr>
<tr>
<td>SQL_MAX_USER_NAME_LEN</td>
<td>SQL_MAXIMUM_USER_NAME_LENGTH</td>
</tr>
<tr>
<td>SQL_MULT_RESULT_SETS</td>
<td>SQL_MULTIPLE_RESULT_SETS</td>
</tr>
<tr>
<td>SQL_OJ_CAPABILITIES</td>
<td>SQL_OUTER_JOIN_CAPABILITIES</td>
</tr>
<tr>
<td>SQL_TXN_CAPABLE</td>
<td>SQL_TRANSACTION_CAPABLE</td>
</tr>
<tr>
<td>SQL_TXN_ISOLATION_OPTION</td>
<td>SQL_TRANSACTION_ISOLATION_OPTION</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**CString Class**

Because objects of the **CString** class in Microsoft® Visual C++® are signed and string arguments in ODBC functions are unsigned, applications that pass **CString** objects to ODBC functions without casting them will receive compiler warnings.
Creating and Terminating Threads

Multithread applications that use ODBC should call the Microsoft® Visual C++® Run-Time Library functions _beginthread and _endthread (or _beginthreadex and _endthreadex) to create and terminate threads that call the ODBC Driver Manager. If applications call the Microsoft Windows NT® functions CreateThread and EndThread instead, memory leaks will occur because the Driver Manager and some ODBC drivers call C run-time functions that will not work on a thread created by calling CreateThread. For more information, see the Microsoft Windows® documentation.

Installing and Configuring the ODBC Software

Note

Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

This section contains the following topics.

- Installing ODBC Components
- Configuring Data Sources
- Installation and Configuration Components Reference
- ODBC Header Files
- Registry Entries for ODBC Components
- Registry Entries for Data Sources

Installing ODBC Components
Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

This section describes how ODBC components are installed and removed. Because driver developers always install an ODBC component (their driver), they need to read this section. Application developers need to read this section only if they will ship ODBC components with their applications. ODBC components include the Driver Manager, drivers, translators, the installer DLL, the cursor library, and any related files. For the purposes of this section, ODBC applications are not considered to be ODBC components.

This section is specific to Microsoft Windows platforms. How ODBC components are installed on other platforms is platform-specific.

ODBC components are installed and removed on a component-by-component basis, not a file-by-file basis. For example, if a translator consists of the translator itself and a number of data files, these files are installed and removed as a group; they must not be installed and removed on a file-by-file basis. The reason for this is to make sure that only complete components exist on the system.

For purposes of installing and removing components, the following are defined to be ODBC components:

- **Core components.** The Driver Manager, cursor library, installer DLL, and any other related files make up the core components and must be installed and removed as a group.
- **Drivers.** Each driver is a separate component.
- **Translators.** Each translator is a separate component.

With the support of Unicode in ODBC 3.5 and later, some consideration must be given to using OLE DB components with ODBC. The 1.1 version of the OLE DB Provider for ODBC was written to specific Unicode specifications within ODBC 3.0. Because these specifications changed in ODBC 3.5, it is necessary to have version 1.5 or later of the provider when using ODBC 3.5 and later. This section contains the following topics.

- Installation Components
- Usage Counting

The installation process starts when the user runs the setup program. The setup program works in conjunction with the installer DLL and a driver setup DLL for each driver. Both the setup program and the installer DLL use the arguments in the `SQLInstallDriverEx` and `SQLInstallTranslatorEx` functions to determine which files to copy or delete for each component. The following illustration shows the relationship between these installation components.
The Odbc.inf file that was used in ODBC 2.x to describe the files required by each ODBC component is not used in ODBC 3.x. Drivers that ship ODBC 3.x components do not need to create an Odbc.inf file. The removal of SQLInstallDriver and SQLInstallODBC, and the deprecation of SQLInstallTranslator, have rendered Odbc.inf unnecessary. The driver information that used to be in the Driver Keyword sections of Odbc.inf is now provided in the lpszDriver argument in SQLInstallDriverEx. The translator information that used to be in the [ODBC Translator] and Translator Specification sections of Odbc.inf is now provided in the lpszTranslator argument of SQLInstallTranslatorEx. These changes allow the ODBC Installer to be more portable across platforms.

For more information about these components, see the following topics at the end of this section.

- Setup Program
- Installer DLL
- Driver Setup DLL

Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

Two types of usage counts are maintained in the registry for each component: a component usage count and one or more optional file usage counts. The component usage count helps the installer DLL maintain registry entries. It is stored in the UsageCount value under the ODBC Core, driver, and translator subkeys. For the format of the UsageCount value and more information about these subkeys, see Registry Entries for ODBC Components.

When a component is first installed, the installer DLL creates a subkey for it and sets the data for the UsageCount value in that subkey to 1. When the component is installed again, the installer DLL increments the usage count. When the component is removed, the installer DLL decrements the usage count. If the usage count falls to 0, the installer DLL removes the subkey for the component.

An application should not physically remove Driver Manager files when the component usage count and the file usage count reach zero.

File usage counts help determine when a file must actually be copied or deleted as opposed to incrementing or decrementing the usage count. This is important because ODBC components, and therefore the files in ODBC components, are shared and can be installed or removed by a variety of applications. The application can delete driver and translator files if the component usage count and the file usage count reach zero. Driver Manager files should not, however, be deleted when both the component usage count and the file usage count have reached zero, because these files can be used by other applications that have not incremented the file usage count.

File usage counts are optional in Microsoft® WindowsNT®/Windows2000.
File usage counts are maintained by the setup program after it calls SQLInstallDriverManager, SQLInstallDriverEx, SQLInstallTranslatorEx, SQLRemoveDriverManager, SQLRemoveDriver, or SQLRemoveTranslator.

When a component is first installed, the setup program or installer DLL creates a value under the following key for each file in that component that is not already on the system:

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>full-path</td>
<td>REG_DWORD</td>
<td>count</td>
</tr>
</tbody>
</table>

It sets the data for those values to 1 and copies the file to the system. When the component is installed again, the setup program or installer DLL increments the usage counts. When the component is removed, the setup program or installer DLL decrements the usage counts. If any usage count falls to 0, the setup program or installer DLL removes the value for the file and, if the component is a driver or a translator, deletes the file. Driver Manager files should not be deleted.

The format of the file usage count value is shown in the following table.

For example, suppose a driver for Informix uses the Infrmx32.dll and Infrmx32.hlp files, and suppose that this driver has been installed twice. The values under the SharedDlls subkey for the Informix driver would be as follows:

© 2016 Microsoft

Configuring Data Sources

Information about data sources is stored in the system registry. Users modify data source information through an administration program. This can be the ODBC Administrator, the ODBC Control Panel device, or an administration program written by an application or driver developer.

You can use PowerShell commands to modify data sources. For more information about these PowerShell commands, see Windows Data Access Components PowerShell Commands.

© 2016 Microsoft
Note

Configuration Components

Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

Data sources are configured by the installer DLL, which in turn calls driver setup DLLs and translator setup DLLs as they are needed. The installer DLL is either invoked directly from Control Panel or loaded and called by another program, known as the administration program. The following illustration shows the relationship between the configuration components.

For more information about these components, see the following topics at the end of this section.

- Setup Program
- Installer DLL
- Driver Setup DLL

See Also

Installation Components

Installation and Configuration Components Reference

Note

Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.
The following sections provide information about the components used to install and configure ODBC.

- Administration Program
- Driver Setup DLL
- Installer DLL
- Setup Program
- Translator Setup DLLs

© 2016 Microsoft

### Administration Program

**Note**

Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

An administration program, the ODBC Administrator, is included with the Windows SDK/MDAC SDK. This program and can be redistributed by users of the SDK. In addition, developers can write their own administration programs. Generally, developers write their own administration programs only if they want to retain complete control over data source configuration, or if they are configuring data sources directly from an application that is acting as an administration program. For example, a spreadsheet program might allow users to add and then use data sources at run time.

The administration program first loads the installer DLL. It then calls functions in the installer DLL to perform the following tasks:

- Add, modify, or delete data sources interactively. The administration program can call SQLManageDataSources, SQLCreateDataSource, or SQLConfigDataSource.

  SQLManageDataSources displays a dialog box with which the user can add, modify, or delete data sources and specify tracing options; this function is called when the installer DLL is invoked directly from the Control Panel. SQLCreateDataSource displays a dialog box with which the user can only add data sources. SQLConfigDataSource passes the call directly to the driver setup DLL.

  In all cases, the installer DLL calls ConfigDSN in the driver setup DLL to actually add, modify, or delete the data source. The driver setup DLL might prompt the user for additional information.

- Add, modify, or delete data sources silently. The administration program calls SQLConfigDataSource in the installer DLL and passes it a null window handle, the name of a data source to add, modify, or delete, and a list of values for the registry. The installer DLL calls ConfigDSN in the driver setup DLL to actually add, modify, or delete the data source.

- Add, modify, or delete a default data source. The default data source is the same as any other data source, except that its name is Default. It is added, modified, or deleted in the same fashion as any other data source.

© 2016 Microsoft

### Driver Setup DLL
Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

The installer DLL contains functions to install and remove ODBC components, maintain registry information about those components, and maintain registry information about data sources. It is written by Microsoft and can be redistributed. For a complete description of the functions in the installer DLL, see Installer DLL API Reference.

See Also

Registry Entries for ODBC Components
Registry Entries for Data Sources

© 2016 Microsoft

Setup Program

Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

© 2016 Microsoft
The user runs the setup program to start the setup process. The setup program is written by the application or driver developer. In addition to installing ODBC components, it can install other software. For example, application developers might use the same setup program both to install ODBC components and to install their applications.

Developers can write the setup program from scratch, using the Microsoft® Windows® SDK setup utilities or setup software from other vendors. This gives those developers complete control over the setup program’s look and feel. The setup program can be written to install additional software, such as an ODBC application. For more information on the Windows SDK setup utilities, see the Windows SDK documentation.

How much of the installation is actually done by the setup program depends on what functions it calls in the installer DLL. The installer DLL contains functions to install individual ODBC components. The setup program simply calls `SQLInstallDriverManager`, `SQLInstallDriverEx`, or `SQLInstallTranslatorEx` in the installer DLL to retrieve the path of the directory in which the component is to be installed and to add information about the component to the registry. These functions do not actually copy files; the setup program does this using the information in the arguments of these functions.

The installer DLL also contains functions to remove ODBC components. The setup program calls `SQLRemoveDriverManager`, `SQLRemoveDriver`, or `SQLRemoveTranslator` in the installer DLL to decrement a component’s usage count in the registry and, if the component’s new usage count falls to 0, remove all information about the component from the registry. These functions do not actually remove the files for the component; the setup program does this if the new usage count falls to 0.

© 2016 Microsoft

### Translator Setup DLLs

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
</table>

Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

The translator setup DLL contains the `ConfigTranslator` function, which returns the default option for a translator. If necessary, it prompts the user for this information. For a complete description of this function, see Setup DLL API Reference.

The translator setup DLL is written by the translator developer. It can be part of the translator DLL or a separate DLL.

© 2016 Microsoft

### ODBC Header Files

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
</table>

Redistribution of ODBC header files is not required. None are considered “Core”.

**Header Files**

- Odbcinst.h
- Sql.h
- Sqlext.h
- Sqltypes.h
Registry Entries for ODBC Components

Note

Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

The installer DLL maintains information in the registry about each installed ODBC component. On computers running Microsoft Windows NT and Microsoft Windows 95/98, this information is stored in subkeys under the following key in the registry:

HKEY_LOCAL_MACHINE
SOFTWARE
ODBC
Odbcinst.ini

Because Odbcinst.ini is a subkey of the HKEY_LOCAL_MACHINE tree, the information about ODBC components is available to all users of the machine.

This section contains the following topics.

- ODBC Core Subkey
- ODBC Drivers Subkey
- Driver Specification Subkeys
- Default Driver Subkey
- ODBC Translators Subkey
- Translator Specification Subkeys

ODBC Core Subkey

The value under the ODBC Core subkey gives the usage count for the core components (Driver Manager, cursor library, installer DLL, and so on). The format of this value is shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
</table>

© 2016 Microsoft
For example, suppose the ODBC Core components have been installed by the setup programs for three different applications and two different drivers. The value under the ODBC Core subkey would be:

```
UsageCount : REG_DWORD : 0x5
```

### ODBC Drivers Subkey

The values under the ODBC Drivers subkey list the installed drivers. The format of these values is shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>driver-description</code></td>
<td>REG_SZ</td>
<td><code>Installed</code></td>
</tr>
</tbody>
</table>

The `driver-description` name is defined by the driver developer. It is usually the name of the DBMS associated with the driver.

For example, suppose drivers have been installed for formatted text files and SQL Server. The values under the ODBC Drivers subkey might be:

```
Text : REG_SZ : Installed
SQL Server : REG_SZ : Installed
```

### Driver Specification Subkeys

Each driver listed in the ODBC Drivers subkey has a subkey of its own. This subkey has the same name as the corresponding value under the ODBC Drivers subkey. The values under this subkey list the full paths of the driver and driver setup DLLs, the values of the driver keywords returned by `SQLDrivers`, and the usage count. The formats of the values are as shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>APILevel</td>
<td>REG_SZ</td>
<td>`0</td>
</tr>
<tr>
<td>ConnectFunctions</td>
<td>REG_SZ</td>
<td>`{Y</td>
</tr>
<tr>
<td>CreateDSN</td>
<td>REG_SZ</td>
<td><code>driver-description</code></td>
</tr>
</tbody>
</table>
The use of each keyword is shown in the following table.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>APILevel</td>
<td>A number indicating the ODBC interface conformance level supported by the driver:</td>
</tr>
<tr>
<td></td>
<td>0 = None</td>
</tr>
<tr>
<td></td>
<td>1 = Level 1 supported</td>
</tr>
<tr>
<td></td>
<td>2 = Level 2 supported</td>
</tr>
<tr>
<td></td>
<td>This must be the same as the value returned for the SQL_ODBC_INTERFACE_CONFORMANCE option in SQLGetInfo.</td>
</tr>
<tr>
<td>CreateDSN</td>
<td>The name of one or more data sources to be created when the driver is installed. The system information must include one data source specification section for each data source listed with the CreateDSN keyword. These sections should not include the Driver keyword, because this is specified in the driver specification section, but must include enough information for the ConfigDSN function in the driver setup DLL to create a data source specification without displaying any dialog boxes. For the format of a data source specification section, see Data Source Specification Subkeys.</td>
</tr>
<tr>
<td>ConnectFunctions</td>
<td>A three-character string indicating whether the driver supports SQLConnect, SQLDriverConnect, and SQLBrowseConnect. If the driver supports SQLConnect, the first character is &quot;Y&quot;; otherwise, it is &quot;N&quot;. If the driver supports SQLDriverConnect, the second character is &quot;Y&quot;; otherwise, it is &quot;N&quot;. If the driver supports SQLBrowseConnect, the third character is &quot;Y&quot;; otherwise, it is &quot;N&quot;. For example, if a driver supports SQLConnect and SQLDriverConnect but not SQLBrowseConnect, the three-character string is &quot;YYN&quot;.</td>
</tr>
<tr>
<td>DriverODBCVer</td>
<td>A character string with the version of ODBC that the driver supports. The version is of the form nn.nn, where the first two digits are the major version and the next two digits are the minor version. For the version of ODBC described in this manual, the driver must return &quot;03.00&quot;. This must be the same as the value returned for the SQL_DRIVER_ODBC_VER option in SQLGetInfo.</td>
</tr>
<tr>
<td>FileExtns</td>
<td>For file-based drivers, a comma-separated list of extensions of the files the driver can use. For example, a dBASE driver might specify *.dbf and a formatted text file driver might specify <em>.txt,</em> .csv. For an example of how an application might use this information, see the FileUsage keyword.</td>
</tr>
</tbody>
</table>
### FileUsage

A number indicating how a file-based driver directly treats files in a data source.

- **0** = The driver is not a file-based driver. For example, an ORACLE driver is a DBMS-based driver.
- **1** = A file-based driver treats files in a data source as tables. For example, an Xbase driver treats each Xbase file as a table.
- **2** = A file-based driver treats files in a data source as a catalog. For example, a Microsoft® Access driver treats each Microsoft Access file as a complete database.

An application might use this to determine how users will select data. For example, Xbase and Paradox users often think of data as stored in files, while ORACLE and Microsoft Access users generally think of data as stored in tables.

When a user selects **Open Data File** from the **File** menu, an application could display the **Windows File Open** common dialog box. The list of file types would use the file extensions specified with the **FileExtns** keyword for drivers that specify a **FileUsage** value of **1** and "Y" as the second character of the value of the **ConnectFunctions** keyword. After the user selects a file, the application would call **SQLDriverConnect** with the **DRIVER** keyword and then execute a **SELECT * FROM table-name** statement.

When the user selects **Import Data** from the **File** menu, an application could display a list of descriptions for drivers that specify a **FileUsage** value of **0** or **2**, and "Y" as the second character of the value of the **ConnectFunctions** keyword. After the user selects a driver, the application would call **SQLDriverConnect** with the **DRIVER** keyword and then display a custom **Select Table** dialog box.

### SQLLevel

A number indicating the SQL-92 grammar supported by the driver:

- **0** = SQL-92 Entry
- **1** = FIPS127-2 Transitional
- **2** = SQL-92 Intermediate
- **3** = SQL-92 Full

This must be the same as the value returned for the **SQL_SQL_CONFORMANCE** option in **SQLGetInfo**.

For information about usage counts, see **Usage Counting** earlier in this section.

Applications should not set the usage count. ODBC will maintain this count.

For example, suppose a driver for formatted text files has a driver DLL named Text.dll, a separate driver setup DLL named Txtsetup.dll, and has been installed three times. If the driver supports the Level 1 API conformance level, supports the Minimum SQL grammar conformance level, treats files as tables, and can use files with the .txt and .csv extensions, the values under the Text subkey might be as follows:

```plaintext
APILevel : REG_SZ : 1
ConnectFunctions : REG_SZ : YYN
Driver : REG_SZ : C:\WINDOWS\SYSTEM32\TEXT.DLL
DriverODBCVer : REG_SZ : 03.00.00
FileExtns : REG_SZ : *.txt,*.csv
FileUsage : REG_SZ : 1
Setup : REG_SZ : C:\WINDOWS\SYSTEM32\TXTSETUP.DLL
SQLLevel : REG_SZ : 0
UsageCount : REG_DWORD : 0x3
```
Default Driver Subkey

The Default subkey contains a single value that describes the driver used by the default data source. The format of this value is shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>REG_SZ</td>
<td>default-driver-description</td>
</tr>
</tbody>
</table>

The `default-driver-description` name is the same as the name of the value under the ODBC Drivers subkey that describes the driver.

For example, if the default data source uses the SQL Server driver, the value under the Default subkey might be:

```
Driver : REG_SZ : SQL Server
```

**Note**

The default driver contained in the Default subkey can refer to either a default user DSN or a default system DSN. If both a default user DSN and a default system DSN have been created, the default driver is determined by the DSN that was created last, so it might not be a valid entry for the DSN that was created first.

ODBC Translators Subkey

The values under the ODBC Translators subkey list the installed translators. The format of these values is shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>translator-desc</td>
<td>REG_SZ</td>
<td>Installed</td>
</tr>
</tbody>
</table>

The `translator-desc` name is defined by the translator developer.

For example, suppose a user has installed the Microsoft® Code Page Translator and a custom ASCII to EBCDIC translator. The values under the ODBC Translators subkey might be as follows:

```
MS Code Page Translator: REG_SZ : Installed
ASCII to EBCDIC: REG_SZ : Installed.
```
Translator Specification Subkeys

Each translator listed in the ODBC Translators subkey has a subkey of its own. This subkey has the same name as the corresponding value under the ODBC Translators subkey. The values under this subkey list the full paths of the translator and translator setup DLLs and the usage count. The formats of the values are as shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translator</td>
<td>REG_SZ</td>
<td>translator-DLL-path</td>
</tr>
<tr>
<td>Setup</td>
<td>REG_SZ</td>
<td>setup-DLL-path</td>
</tr>
<tr>
<td>UsageCount</td>
<td>REG_DWORD</td>
<td>count</td>
</tr>
</tbody>
</table>

For information about usage counts, see Usage Counting earlier in this section.

Applications should not set the usage count. ODBC will maintain this count.

For example, suppose the Microsoft Code Page Translator has a translation DLL named Mscpxl32.dll, that the translator setup functions are in the same DLL, and that the translator has been installed three times. The values under the Microsoft Code Page Translator subkey might be as follows:

```
Translator : REG_SZ : C:\WINDOWS\SYSTEM32\MSCPX32.DLL
Setup : REG_SZ : C:\WINDOWS\SYSTEM32\MSCPX32.DLL
UsageCount : REG_DWORD : 0x3
```

Registry Entries for Data Sources

Note

Starting with Windows XP and Windows Server 2003, ODBC is included in the Windows operation system. You should only explicitly install ODBC on earlier versions of Windows.

The installer DLL maintains information in the registry about each data source. In Microsoft Windows NT/Windows 2000 and Microsoft Windows 95/98, this information is stored in subkeys under one of the following two keys in the registry:

- HKEY_LOCAL_MACHINE
  - SOFTWARE
    - ODBC
      - Odbc.ini
  - HKEY_CURRENT_USER
Which key is used depends on whether the data source is a **system data source**, which is available to all users, or a **user data source**, which is available only to the current user. System data sources are stored on the HKEY_LOCAL_MACHINE tree, and user data sources are stored on the HKEY_CURRENT_USER tree. In all other respects, system data sources and user data sources are identical.

This section contains the following topics.

- **ODBC Data Sources Subkey**
- **Data Source Specification Subkeys**
- **Default Subkey**
- **ODBC Subkey**

---

### ODBC Data Sources Subkey

The values under the ODBC Data Sources subkey list the data sources. The format of these values is as shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-source-name</td>
<td>REG_SZ</td>
<td>driver-description</td>
</tr>
</tbody>
</table>

The `data-source-name` value is defined by the administration program (which usually prompts the user for it), and `driver-description` is defined by the driver developer (it is usually the name of the DBMS associated with the driver).

For example, suppose three data sources have been defined: Inventory, which uses SQL Server; Payroll, which uses dBASE; and Personnel, which uses formatted text files. The values under the ODBC Data Sources subkey might be as follows:

```
Inventory : REG_SZ : SQL Server
Payroll : REG_SZ : dBASE
Personnel : REG_SZ : Text
```

---

### Data Source Specification Subkeys

Each data source listed in the ODBC Data Sources subkey has a subkey of its own. This subkey has the same name as the corresponding value under the ODBC Data Sources subkey. The values under this subkey must list the driver DLL and may list a description of the data source. If the driver supports translators, the values may list the name of
a default translator, the default translation DLL, and the default translation option. The values may also list other information required by the driver to connect to the data source. For example, the driver might require a server name, database name, or schema name.

The formats of the values are as shown in the following table. Only the Driver value is required.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>REG_SZ</td>
<td>description</td>
</tr>
<tr>
<td>Driver</td>
<td>REG_SZ</td>
<td>driver-DLL-path</td>
</tr>
<tr>
<td>TranslationDLL</td>
<td>REG_SZ</td>
<td>translator-DLL-path</td>
</tr>
<tr>
<td>TranslationName</td>
<td>REG_SZ</td>
<td>translator-name</td>
</tr>
<tr>
<td>TranslationOption</td>
<td>REG_SZ</td>
<td>translation-option</td>
</tr>
<tr>
<td>opt-value-name</td>
<td></td>
<td>opt-value-type</td>
</tr>
<tr>
<td>opt-value-data</td>
<td></td>
<td>opt-value-data</td>
</tr>
</tbody>
</table>

For example, suppose the SQL Server driver requires the server name and a flag for OEM to ANSI conversion and defines the Server and OEMTOANSI values for these. Suppose also that the Inventory data source uses the Microsoft® Code Page Translator to translate between the Windows® Latin 1 (1250) and Multilingual (850) code pages. The values under the Inventory subkey might be as follows:

```plaintext
Description : REG_SZ : Inventory database on server InvServ
Driver : REG_SZ : C:\WINDOWS\SYSTEM32\SQLSRV32.DLL
OEMTOANSI : REG_SZ : Yes
Server : REG_SZ : InvServ
TranslationDLL : REG_SZ : C:\WINDOWS\SYSTEM32\MSCPX32.DLL
TranslationName : REG_SZ : MS Code Page Translator
TranslationOption : REG_SZ : 12500850
```

© 2016 Microsoft

**Default Subkey**

The registry may specify a default data source with the Default subkey. This subkey is a special case of a data source specification subkey and has the same values as any other data source specification subkey. The only difference is that it is not listed as a value under the ODBC Data Sources subkey.

© 2016 Microsoft
ODBC Subkey

The values under the ODBC subkey specify ODBC tracing options. These options are set through the Tracing tab of the ODBC Data Source Administrator dialog box displayed by SQLManageDataSources. The ODBC subkey itself is optional. The format of these values is as shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace</td>
<td>REG_SZ</td>
<td>0</td>
</tr>
<tr>
<td>TraceFile</td>
<td>REG_SZ</td>
<td>tracefile-path</td>
</tr>
</tbody>
</table>

The values have the meanings described in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| Trace   | If the Trace value is set to 1 when an application calls SQLAllocHandle with the SQL_HANDLE_ENV option, tracing is enabled for the calling application.  
If the Trace keyword is set to 0 when an application calls SQLAllocHandle with the SQL_HANDLE_ENV option, tracing is disabled for the calling application. This is the default value.  
An application can enable or disable tracing with the SQL_ATTR_TRACE connection attribute. However, doing so does not change the data for this value. |
| TraceFile| If tracing is enabled, the Driver Manager writes to the trace file specified by the TraceFile value.  
If no trace file is specified, the Driver Manager writes to the Sql.log file on the current drive. This is the default value.  
Tracing should be used only for a single application, or each application should specify a different trace file. Otherwise, two or more applications will attempt to open the same trace file at the same time, causing an error.  
An application can specify a new trace file with the SQL_ATTR_TRACEFILE connection attribute. However, doing so does not change the data for this value. |

For example, suppose that tracing is enabled and the trace file is C:\Odbc.log. The values under the ODBC subkey would be as follows:

Trace : REG_SZ : 1  
TraceFile : REG_SZ : C:\ODBC.LOG

Developing an ODBC Driver
This section contains topics that discuss how to develop an ODBC driver.

This section contains the following topics:

- ODBC Driver Architecture
- Upgrading a 3.5 Driver to a 3.8 Driver
- Developing Connection-Pool Awareness in an ODBC Driver
- Notification of Asynchronous Function Completion

© 2016 Microsoft

# ODBC Driver Architecture

Driver writers must be aware that the driver architecture can affect whether an application can use DBMS-specific SQL.

![Diagram showing ODBC Driver Architecture]

**File-based Drivers**

When the driver accesses the physical data directly, the driver acts as both driver and data source. The driver must process both ODBC calls and SQL statements. Developers of file-based drivers must write their own database engines.

**DBMS-Based Drivers**

When a separate database engine is used to access physical data, the driver processes only ODBC calls. It passes SQL statements to the database engine for processing.
Network Architecture

File and DBMS ODBC configurations can exist on a single network.

Other Driver Architectures

When a driver is required to work with a variety of data sources, it can be used as middleware. Heterogeneous join engine architecture can make the driver appear as a driver manager. Drivers can also be installed on servers, where they can be shared by a series of clients.

For more information about driver architecture, see Driver Manager and Driver Architecture in the section on ODBC Architecture.

More information about driver issues can be found in the locations described in the following table.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Topic</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compatibility issues with applications and drivers</td>
<td>Application/Driver Compatibility</td>
<td>Programming Considerations, in the ODBC Programmer’s Reference</td>
</tr>
<tr>
<td>Writing ODBC drivers</td>
<td>Writing ODBC 3.x Drivers</td>
<td>Programming Considerations, in the ODBC Programmer’s Reference</td>
</tr>
<tr>
<td>Driver guidelines for backward compatibility</td>
<td>Driver Guidelines for Backward Compatibility</td>
<td>Appendix G: Driver Guidelines for Backward Compatibility, in the ODBC Programmer’s Reference</td>
</tr>
<tr>
<td>Connecting to a driver</td>
<td>Choosing a Data Source or Driver</td>
<td>Connecting to a Data Source or Driver, in the ODBC Programmer’s Reference</td>
</tr>
<tr>
<td>Identifying drivers</td>
<td>Viewing Drivers</td>
<td>Viewing Drivers, in the Microsoft ODBC Data Source Administrator online Help</td>
</tr>
<tr>
<td>Enabling connection pooling</td>
<td>ODBC Connection Pooling</td>
<td>Connecting to a Data Source or Driver, in the ODBC Programmer’s Reference</td>
</tr>
<tr>
<td>Unicode/ANSI driver and connection issues</td>
<td>Unicode Drivers</td>
<td>Programming Considerations, in the ODBC Programmer’s Reference</td>
</tr>
</tbody>
</table>

See Also

Developing an ODBC Driver

© 2016 Microsoft

Upgrading a 3.5 Driver to a 3.8 Driver

This topic provides guidelines and considerations for upgrading an ODBC 3.5 driver to an ODBC 3.8 driver.

Version Numbers

The following guidelines relate to version numbers:
A driver should support SQL_OV_ODBC3_80 for SQL_ATTR_ODBC_VERSION, returning SQL_ERROR for values other than SQL_OV_ODBC2, SQL_OV_ODBC3, and SQL_OV_ODBC3_80. Future versions of the Driver Manager will assume that a driver supports an ODBC compliance level if the driver returns SQL_SUCCESS from SQLSetEnvAttr Function.

A version 3.8 driver should return 03.80 from SQLGetInfo when SQL_DRIVER_ODBC_VER is passed to InfoType. However, older Driver Managers, which were included in older versions of Microsoft Windows, will treat the driver as a version 3.5 driver, and issue a warning.

In Windows 7, the Driver Manager version is 03.80. In Windows 8, the Driver Manager version is 03.81 via the SQLGetInfo SQL_DM_VER (InfoType parameter). SQL_ODBC_VER reports the version as 03.80 in both Windows 7 and Windows 8.

Driver–Specific C Data Types

A driver can have customized C data types when it works with a version 3.8 ODBC application. (For more information, see C Data Types in ODBC.) However, there is no requirement for a 3.8 driver to implement any driver-specific C types. But the driver should still perform the range check of C types; the Driver Manager will not do that for 3.8 drivers. To facilitate driver development, the value of the driver specific, C data type can be defined in the following format:

```
SQL_DRIVER_C_TYPE_BASE+0, SQL_DRIVER_C_TYPE_BASE+1
```

Driver–specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes

When developing a new driver, you should use the driver-specific range for data types, descriptor types, information types, diagnostic types, and attributes. Driver-specific ranges and their base type values are discussed in Driver–Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes.

Connection Pooling

For better management of connection pooling, ODBC 3.8 introduces the SQL_ATTR_RESET_CONNECTION connection attribute in SQLSetConnectAttr. SQL_RESET_CONNECTION_YES is the only valid value for this attribute. SQL_ATTR_RESET_CONNECTION will be set just before the Driver Manager puts a connection in the connection pool, allowing the driver to reset the other connection attributes to their default values.

To avoid unnecessary communication with the server, a driver can defer the connection attribute reset until the next communication with the remote server, after the connection is reused from the pool.

Note that SQL_ATTR_RESET_CONNECTION is only used in communication between the Driver Manager and a driver. An application cannot set this attribute directly. All version 3.8 drivers should implement this connection attribute.

Streamed Output Parameters

ODBC version 3.8 introduces streamed output parameters, a more scalable way to retrieve output parameters. (For more information, see Retrieving Output Parameters Using SQLGetData.) To support this feature, a driver
should set SQL_GD_OUTPUT_PARAMS in the return value when SQL_GETDATA_EXTENSIONS is the InfoType in a SQLGetInfo call. Support for an SQL type with streamed output parameters must be implemented in the driver. The Driver Manager will not generate an error for an invalid SQL type. The SQL types that support streamed output parameters is defined in the driver.

A driver should return SQL_ERROR if the application used SQLGetData to retrieve a parameter that is not the same as the parameter returned by SQLParamData.

Asynchronous Execution for Connection Operations (Polling Method)

A driver can enable asynchronous support for various connection operations.

Beginning in Windows 7, ODBC supports the polling method (for more information, see Asynchronous Execution (Polling Method)). There is no requirement for a version 3.8 ODBC driver to implement asynchronous operations on connection handles. Even if a driver does not implement asynchronous operations on connection handles, the driver should still implement the SQL_ASYNC_DBC_FUNCTIONS InfoType and return SQL_ASYNC_DBC_NOT_CAPABLE.

When asynchronous connection operations are enabled, the running time of a connection operation is the total time of all repeated calls. If the last repeated call occurs after the total time has exceeded the value set by the SQL_ATTR_CONNECTION_TIMEOUT connection attribute, and the operation has not finished, the driver returns SQL_ERROR and logs a diagnostic record with SQLState HYT01 and the message "Connection timeout expired". There is no timeout if the operation finished.

SQL Cancel Handle Function

ODBC 3.8 supports SQLCancelHandle Function, which is used to cancel both connection and statement operations. A driver that supports SQLCancelHandle must export the function. A driver should not cancel any synchronous or asynchronous connection function that is in progress if the application calls SQLCancel or SQLCancelHandle on a statement handle. Similarly, a driver should not cancel any synchronous or asynchronous statement function that is in progress if an application calls SQLCancelHandle on the connection handle. Also, a driver should not cancel the browsing operation (SQLBrowseConnect returns SQL_NEED_DATA) if the application calls SQLCancelHandle on the connection handle. In these cases, a driver should return HY010, "function sequence error".

It is not necessary to support both SQLCancelHandle and asynchronous connection operations at the same time. A driver can support asynchronous connection operations but not SQLCancelHandle, or vice versa.

Suspended Connections

The ODBC 3.8 Driver Manager can put a connection into suspended state. An application will call SQLDisconnect to release resources associated with the connection. In this case, a driver should try to release as many resources as possible without checking the state of the connection. For more information about the suspended state, see SQLEndTran Function.

Driver–Aware Connection Pooling

ODBC in Windows 8 allows drivers to customize connection pool behavior. For more information, see Driver-Aware Connection Pooling.
Developing Connection-Pool Awareness in an ODBC Driver

This topic discusses the details of developing an ODBC driver that contains information about how the driver should provide connection pooling services.

Enabling Driver-Aware Connection Pooling

A driver must implement the following ODBC Service Provider Interface (SPI) functions:

- SQLSetConnectAttrForDbcInfo
- SQLSetConnectInfo
- SQLSetDriverConnectInfo
- SQLGetPoolID
- SQLRateConnection
- SQLPoolConnect
- SQLCleanupConnectionPoolID

See ODBC Service Provider Interface (SPI) Reference for more information.

A driver must also implement the following existing functions so that the driver-aware pooling can be enabled:

<table>
<thead>
<tr>
<th>Function</th>
<th>Added Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocHandle</td>
<td>Support the new handle type: SQL_HANDLE_DBC_INFO_TOKEN (see the description below).</td>
</tr>
<tr>
<td>SQLFreeHandle</td>
<td></td>
</tr>
<tr>
<td>SQLGetDiagField</td>
<td></td>
</tr>
<tr>
<td>SQLGetDiagRec</td>
<td></td>
</tr>
</tbody>
</table>
Support the new set-only connection attribute: SQL_ATTR_DBC_INFO_TOKEN for resetting the connection (see the description below).

Note

Deprecated functions such as SQLError and SQLSetConnectOption are not supported for driver-aware connection pooling.

The Pool ID

The pool ID is a pointer-length driver-specific ID to represent a particular group of connections that can be used interchangeably. Given a set of connection information, a driver should be able to quickly deduce the corresponding pool ID.

For example, the pool ID should encode the server name and credential information. However, the database name is not needed because a driver may be able to reuse a connection and then change the database in less time than making a new connection.

A driver should define a set of key attributes, which will comprise the pool ID. The value of these key attributes can come from connection attributes, connection string, and DSN. In case there are any conflicts in these sources, the existing, driver-specific resolution policy should be used for backward compatibility.

The Driver Manager will use a different pool for different pool IDs. All connections in the same pool are reusable. The Driver Manager will never reuse a connection with a different pool ID.

Therefore drivers should assign a unique pool ID for every group of connections with the same value in their defined key attributes. If a driver uses the same pool ID for two connections with different values in their key attributes, the Driver Manager will still put them into the same pool (the Driver Manager knows nothing about the driver-specific key attributes). This means that the driver will need to report to the Driver Manager that a connection with a different set of key attributes is not reusable inside SQLRateConnection Function. This can decrease performance and this is not recommended.

The Driver Manager will not reuse a connection allocated from another driver environment even if all connection information matches. The Driver Manager will use a different pool for different environment, even when connections have the same pool ID. Therefore, the pool ID is local to its driver environment.

The function for getting the pool ID from the driver is SQLGetPoolID Function.

The Connection Rating

Compared to establishing a new connection, you can get better performance by resetting some connection information (such as DATABASE) in a pooled connection. So, you may not want the database name to be in your set of key attributes. Otherwise, you can have a separate pool for each database, which may not be good in mid-tier applications, where customers use various different connection strings.

Whenever you reuse a connection that has some attribute mismatch, you should reset the mismatched attributes based on the new application request, so that the returned connection is identical to the application request (see the discussion of the attribute SQL_ATTR_DBC_INFO_TOKEN in SQLSetConnectAttr Function). However, resetting those attributes may decrease performance. For example, resetting a database requires a network call to server. Therefore, reuse a connection that is perfectly matched, if one is available.

A rating function in the driver can evaluate an existing connection with a new connection request. For example, the driver’s rating function can determine:

- If the existing connection is perfectly matched with the request.
- If there are only some insignificant mismatches, such as connection timeout, which do not require
communication with the server to reset.

- If there are some mismatched attributes that require a communication with the server to reset but would still result in better performance than establishing a new connection.

- If the mismatched occurred for an attribute that is very time-consuming to reset (the developer of the driver may consider adding this attribute into the set of key attributes, which is used to generate the pool ID).

A score between 0 and 100 is possible, where 0 means do not reuse and 100 means perfectly matched. SQLRateConnection is the function for rating a connection.

## New ODBC Handle – SQL_HANDLE_DBC_INFO_TOKEN

To support driver-aware connection pooling, the driver needs connection information to compute the Pool ID. The driver also needs connection information to compare new connection requests with connections in the pool. Whenever no connection in the pool can be reused, the driver has to establish a new connection, hence requiring connection information.

Since connection information can come from multiple sources (connection string, connection attributes, and DSN), the driver may need to parse the connection string and resolve the conflict between these sources in each of the above function call.

Therefore, a new ODBC handle is introduced: SQL_HANDLE_DBC_INFO_TOKEN. With SQL_HANDLE_DBC_INFO_TOKEN, a driver does not need to parse the connection string and resolve conflicts in connection information more than once. Since this is a driver-specific data structure, the driver can store data such as connection information or pool ID.

This handle is only used as an interface between the Driver Manager and driver. An application cannot allocate this handle directly.

The parent handle of this handle is of type SQL_HANDLE_ENV, meaning that the driver can obtain the environment information from the HENV handle during connection information resolution.

Whenever it receives a new connection request, the Driver Manager will allocate a handle of type SQL_HANDLE_DBC_INFO_TOKEN for storing connection information, after it confirms that the driver supports connection-pool awareness. When finished using the handle (but before returning some return codes other than SQL_STILL_EXECUTING from SQLDriverConnect or SQLConnect), the Driver Manager will free the handle. Therefore, the handle is created after the SQLAllocHandle call, and destroyed after the SQLFreeHandle call. The Driver Manager guarantees the handle will be freed before freeing its associated HENV (when SQLDriverConnect or SQLConnect returns an error).

The driver should modify the following functions to accept the new handle type SQL_HANDLE_DBC_INFO_TOKEN:

1. SQLAllocHandle
2. SQLFreeHandle
3. SQLGetDiagField
4. SQLGetDiagRec

The Driver Manager guarantees that multiple threads will not use the same SQL_HANDLE_DBC_INFO_TOKEN handle simultaneously. Therefore, the synchronization model of this handle can be very simple inside the driver. The Driver Manager will not take an environment lock before allocating and freeing SQL_HANDLE_DBC_INFO_TOKEN.

The Driver Manager’s SQLAllocHandle and SQLFreeHandle will not accept this new handle type.

SQL_HANDLE_DBC_INFO_TOKEN may contain confidential information such as credentials. Therefore, a driver
should securely clear the memory buffer (using SecureZeroMemory) that contains the sensitive information before releasing this handle with SQLFreeHandle. Whenever an application’s environment handle is closed, all associated connection pools will be closed.

Driver Manager Connection Pool Rating Algorithm

This section discusses the rating algorithm for Driver Manager connection pooling. Driver developers can implement the same algorithm for backward compatibility. This algorithm may not be the best one. You should refine this algorithm based on your implementation (otherwise, there is no reason to implement this feature).

The Driver Manager will return an integral rating from 0 to 100 for each connection from the pool. 0 means the connection cannot be reused and 100 indicates a perfect match. Assume the connection request is named hRequest and the existing connection from the pool is named as hCandidate. If any one of the following conditions is false, the pooled connection hCandidate cannot be reused for hRequest (the Driver Manager will assign a rating of 0).

- hCandidate and hRequest both come from either UNICODE API (such as SQLDriverConnectW) or ANSI API (such as SQLDriverConnectA). (UNICODE drivers can behave differently given ANSI API and UNICODE API (see the connection attribute SQL_ATTR_ANSI_APP).)

- hCandidate and hRequest are created by the same function; either SQLDriverConnect or SQLConnect.

- The connection string used to open hCandidate should be the same as hRequest, when SQLDriverConnect is used.

- The ServerName (or DSN), user name, and password used to open hCandidate should be the same used to open hRequest when SQLConnect is used.

- The security identifier (SID) of the current thread should be the same as the SID used to open hCandidate.

- For driver that is expensive to enlist and unenlist (see the discussion of SQL_DTC_TRANSITION_COST in SQLConnect), reusing hRequest must not require an extra enlistment or unenlistment.

The following table shows score assignment for different scenarios.

<table>
<thead>
<tr>
<th>Comparison on connection attributes between the pooled connection and the request</th>
<th>No Enlistment / unenlistment</th>
<th>Require Extra Enlistment / Unenlistment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalog ( SQL_ATTR_CURRENT_CATALOG ) is different</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>Some connection attributes are different, but catalog is the same</td>
<td>90</td>
<td>70</td>
</tr>
<tr>
<td>All connection attributes perfectly matched</td>
<td>100</td>
<td>80</td>
</tr>
</tbody>
</table>

Sequence Diagram

This sequence diagram shows the basic pooling mechanism described in this topic. It only shows the use of SQLDriverConnect but the SQLConnect case is similar.
State Diagram

This state diagram shows the connection info token object, described in this topic. The diagram only shows
SQLDriverConnect but the SQLConnect case is similar. Since the Driver Manager may need to handle errors at any time, the Driver Manager can call SQLFreeHandle for any state.

See Also

Driver-Aware Connection Pooling
ODBC Service Provider Interface (SPI) Reference

© 2016 Microsoft

Notification of Asynchronous Function Completion
In the Windows 8 SDK, ODBC added a mechanism to notify applications when an asynchronous operation completes, which we will refer to as "notification on completion". (See Asynchronous Execution (Notification Method) for more information.) This topic discusses some of the issues for driver developers.

The Interface between the Driver Manager and Driver

The Driver Manager internally provides a callback function `SQLAsyncNotificationCallback Function`. `SQLAsyncNotificationCallback` can only be called by the driver -- an application cannot directly call it. The driver calls `SQLAsyncNotificationCallback` whenever new data received from the server after last returning `SQL_STILL_EXECUTING`.

The Driver Manager provides a callback mechanism so a driver can notify the Driver Manager when some progress has been made in executing an asynchronous operation after the corresponding function returns `SQL_STILL_EXECUTING`. The Driver Manager sets the `SQL_ATTR_ASYNC_DBC_NOTIFICATION_CALLBACK` attribute on a driver connection handle with a non-NULL function pointer, which is of type `SQL_ASYNC_NOTIFICATION_CALLBACK`, for the driver to work in notification mode for any asynchronous operations on that handle. Similarly, the Driver Manager sets the `SQL_ATTR_ASYNC_STMT_NOTIFICATION_CALLBACK` attribute on a driver statement handle with a non-NULL function pointer, which is also of type `SQL_ASYNC_NOTIFICATION_CALLBACK`, for the driver to work in notification mode for any asynchronous operations on that handle.

If an asynchronous operation is performed on a driver handle, the asynchronous driver functions should work in a non-blocking style. If the operation cannot complete immediately, the driver function should return `SQL_STILL_EXECUTING`. This requirement is true for both polling mode and notification mode.

If a handle is in notification asynchronous mode, the driver must call the notification callback function, whose address is the value for the `SQL_ATTR_ASYNC_DBC_NOTIFICATION_CALLBACK` or `SQL_ATTR_ASYNC_STMT_NOTIFICATION_CALLBACK` attribute, once after returning `SQL_STILL_EXECUTING`. In other words, one returning `SQL_STILL_EXECUTING` must be paired with one invocation of the notification callback function. The driver should use the current value of the `SQL_ATTR_ASYNC_DBC_NOTIFICATION_CONTEXT` or `SQL_ATTR_ASYNC_STMT_NOTIFICATION_CONTEXT` handle attribute as the value for the call back function parameter `pContext`.

The driver must not call back in the thread that calls the driver function; there is no reason to notify progress before the function returns. The driver should use its own thread to callback. The Driver Manager will not use the driver's callback thread for executing extensive processing logic.

The Driver Manager will call the original function again after the driver calls back. The Driver Manager may use a thread that is neither an application thread nor a driver thread. If the driver uses some information associated with the thread (for example, security token or user identifier), the driver should save the required information in the initial asynchronous call and use the saved value before the whole asynchronous operation completes. Usually, only `SQLDriverConnect`, `SQLConnect`, or `SQLBrowseConnect` need to use that kind of information.

See Also

Developing an ODBC Driver

© 2016 Microsoft

SQLAsyncNotificationCallback Function

Conformance
Summary

`SQLAsyncNotificationCallback` allows a driver to call back to the Driver Manager when there is some progress for the current asynchronous operation after the driver returns `SQL_STILL_EXECUTING`. `SQLAsyncNotificationCallback` can only be called by the driver.

Drivers do not call `SQLAsyncNotificationCallback` with function name `SQLAsyncNotificationCallback`. Instead, the Driver Manager passes a function pointer to a driver as the value for the `SQL_ATTR_ASYNC_DBC_NOTIFICATION_CALLBACK` or `SQL_ATTR_ASYNC_STMT_NOTIFICATION_CALLBACK` attribute of the corresponding connection handle or statement handle, respectively. Different handles may be assigned different function pointer values. The type of the function pointer is defined as `SQL_ASYNC_NOTIFICATION_CALLBACK`.

`SQLAsyncNotificationCallback` is thread-safe. A driver can choose to use multiple threads calling `SQLAsyncNotificationCallback` on different handles simultaneously.

Syntax

```c
typedef SQLRETURN (SQL_API *SQL_ASYNC_NOTIFICATION_CALLBACK)(
    SQLPOINTER pContex,
    BOOL fLast);
```

Arguments

- **pContex**
  
  Pointer to a data structure defined by the Driver Manager. The value is passed to the driver via `SQLSetConnectAttr` with `SQL_ATTR_ASYNC_DBC_NOTIFICATION_CONTEXT` or `SQLSetStmtAttr` with `SQL_ATTR_ASYNC_STMT_NOTIFICATION_CONTEXT`. The driver does not have access to the value.

- **fLast**
  
  Used by a driver to indicate that this callback function invocation is the last one for the current asynchronous operation. The driver will return a return code other than `SQL_STILL_EXECUTING` when the Driver Manager calls the function again. The Driver Manager may use this information, for example, to inform the application in advance that the asynchronous operation will complete.

If `Handle` is not a valid handle of the type specified by `HandleType`, `SQLCancelHandle` returns `SQL_INVALID_HANDLE`.

Returns

`SQL_SUCCESS` or `SQL_ERROR`.

Diagnostics

`SQLAsyncNotificationCallback` can return `SQL_ERROR` for the following two situations (these indicate an implementation problem in the driver or Driver Manager):
<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection or statement did not request notification.</td>
<td></td>
</tr>
<tr>
<td>Invalid handle</td>
<td>The driver passed in an invalid handle, which failed the internal Driver Manager validation tests.</td>
</tr>
</tbody>
</table>

See Also

Asynchronous Execution (Polling Method)
ODBC Reference

The following topics contain syntax and semantic information for all ODBC functions.

- Function Summary
- ODBC API Reference
- Setup DLL API Reference
- Installer DLL API Reference
- Translation DLL Function Reference
- ODBC Service Provider Interface (SPI) Reference

© 2016 Microsoft

Function Summary

The following sections summarize the functions used by ODBC-enabled applications and related software.

- ODBC Function Summary
- Setup DLL Function Summary
- Installer DLL Function Summary
- Translation DLL Function Summary
- ODBC Service Provider Interface Summary

© 2016 Microsoft

ODBC Function Summary

The following table lists ODBC functions, grouped by type of task, and includes the conformance designation and a brief description of the purpose of each function. For more information about conformance designations, see ODBC and the Standard CLI. For more information about the syntax and semantics for each function, see ODBC API Reference.
An application can call the **SQLGetInfo** function to obtain conformance information about a driver. To obtain information about support for a specific function in a driver, an application can call **SQLGetFunctions**.

<table>
<thead>
<tr>
<th>Task</th>
<th>Function name</th>
<th>Conformance</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connecting to a data source</td>
<td>SQLAllocHandle</td>
<td>ISO 92</td>
<td>Obtains an environment, connection, statement, or descriptor handle.</td>
</tr>
<tr>
<td></td>
<td>SQLConnect</td>
<td>ISO 92</td>
<td>Connects to a specific driver by data source name, user ID, and password.</td>
</tr>
<tr>
<td></td>
<td>SQLDriverConnect</td>
<td>ODBC</td>
<td>Connects to a specific driver by connection string or requests that the Driver Manager and driver display connection dialog boxes for the user.</td>
</tr>
<tr>
<td></td>
<td>SQLBrowseConnect</td>
<td>ODBC</td>
<td>Returns successive levels of connection attributes and valid attribute values. When a value has been specified for each connection attribute, connects to the data source.</td>
</tr>
<tr>
<td>Obtaining information about a driver and data source</td>
<td>SQLDataSources</td>
<td>ISO 92</td>
<td>Returns the list of available data sources.</td>
</tr>
<tr>
<td></td>
<td>SQLDrivers</td>
<td>ODBC</td>
<td>Returns the list of installed drivers and their attributes.</td>
</tr>
<tr>
<td></td>
<td>SQLGetInfo</td>
<td>ISO 92</td>
<td>Returns information about a specific driver and data source.</td>
</tr>
<tr>
<td></td>
<td>SQLGetFunctions</td>
<td>ISO 92</td>
<td>Returns supported driver functions.</td>
</tr>
<tr>
<td></td>
<td>SQLGetTypeInfo</td>
<td>ISO 92</td>
<td>Returns information about supported data types.</td>
</tr>
<tr>
<td>Setting and retrieving driver attributes</td>
<td>SQLSetConnectAttr</td>
<td>ISO 92</td>
<td>Sets a connection attribute. Returns the value of a connection attribute.</td>
</tr>
<tr>
<td></td>
<td>SQLGetConnectAttr</td>
<td>ISO 92</td>
<td>Sets an environment attribute.</td>
</tr>
<tr>
<td></td>
<td>SQLSetEnvAttr</td>
<td>ISO 92</td>
<td>Returns the value of an environment attribute.</td>
</tr>
<tr>
<td>Function</td>
<td>Specification</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>SQLSetStmtAttr</td>
<td>ISO 92</td>
<td>Sets a statement attribute.</td>
<td></td>
</tr>
<tr>
<td>SQLGetStmtAttr</td>
<td>ISO 92</td>
<td>Returns the value of a statement attribute.</td>
<td></td>
</tr>
<tr>
<td>SQLGetDescField</td>
<td>ISO 92</td>
<td>Returns the value of a single descriptor field.</td>
<td></td>
</tr>
<tr>
<td>SQLGetDescRec</td>
<td>ISO 92</td>
<td>Returns the values of multiple descriptor fields.</td>
<td></td>
</tr>
<tr>
<td>SQLSetDescField</td>
<td>ISO 92</td>
<td>Sets a single descriptor field.</td>
<td></td>
</tr>
<tr>
<td>SQLSetDescRec</td>
<td>ISO 92</td>
<td>Sets multiple descriptor fields.</td>
<td></td>
</tr>
<tr>
<td>SQLCopyDesc</td>
<td>ISO 92</td>
<td>Copies descriptor information from one descriptor handle to another.</td>
<td></td>
</tr>
<tr>
<td>SQLPrepare</td>
<td>ISO 92</td>
<td>Prepares an SQL statement for later execution.</td>
<td></td>
</tr>
<tr>
<td>SQLBindParameter</td>
<td>ODBC</td>
<td>Assigns storage for a parameter in an SQL statement.</td>
<td></td>
</tr>
<tr>
<td>SQLGetCursorName</td>
<td>ISO 92</td>
<td>Returns the cursor name associated with a statement handle.</td>
<td></td>
</tr>
<tr>
<td>SQLSetCursorName</td>
<td>ISO 92</td>
<td>Specifies a cursor name.</td>
<td></td>
</tr>
<tr>
<td>SQLSetScrollOptions</td>
<td>ODBC</td>
<td>Sets options that control cursor behavior.</td>
<td></td>
</tr>
<tr>
<td>SQLExecute</td>
<td>ISO 92</td>
<td>Executes a prepared statement.</td>
<td></td>
</tr>
<tr>
<td>SQLExecDirect</td>
<td>ISO 92</td>
<td>Executes a statement.</td>
<td></td>
</tr>
<tr>
<td>SQLNativeSql</td>
<td>ODBC</td>
<td>Returns the text of an SQL statement as translated by the driver.</td>
<td></td>
</tr>
<tr>
<td>SQLDescribeParam</td>
<td>ODBC</td>
<td>Returns the description for a specific parameter in a statement.</td>
<td></td>
</tr>
<tr>
<td>SQLNumParams</td>
<td>ISO 92</td>
<td>Returns the number of parameters in a statement.</td>
<td></td>
</tr>
<tr>
<td>SQLParamData</td>
<td>ISO 92</td>
<td>Used in conjunction with SQLPutData to supply parameter data at execution time. (Useful for long data values.)</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Standard</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>SQLPutData</td>
<td>ISO 92</td>
<td>Sends part or all of a data value for a parameter. (Useful for long data values.)</td>
<td></td>
</tr>
<tr>
<td>Retrieving results and information about results</td>
<td>SQLRowCount</td>
<td>Returns the number of rows affected by an insert, update, or delete request.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQLNumResultCols</td>
<td>Returns the number of columns in the result set.</td>
<td></td>
</tr>
<tr>
<td>SQLDescribeCol</td>
<td>ISO 92</td>
<td>Describes a column in the result set.</td>
<td></td>
</tr>
<tr>
<td>SQLColAttribute</td>
<td>ISO 92</td>
<td>Describes attributes of a column in the result set.</td>
<td></td>
</tr>
<tr>
<td>SQLBindCol</td>
<td>ISO 92</td>
<td>Assigns storage for a result column and specifies the data type.</td>
<td></td>
</tr>
<tr>
<td>SQLFetch</td>
<td>ISO 92</td>
<td>Returns multiple result rows.</td>
<td></td>
</tr>
<tr>
<td>SQLFetchScroll</td>
<td>ISO 92</td>
<td>Returns scrollable result rows.</td>
<td></td>
</tr>
<tr>
<td>SQLGetData</td>
<td>ISO 92</td>
<td>Returns part or all of one column of one row of a result set. (Useful for long data values.)</td>
<td></td>
</tr>
<tr>
<td>SQLSetPos</td>
<td>ODBC</td>
<td>Positions a cursor within a fetched block of data and allows an application to refresh data in the rowset or to update or delete data in the result set.</td>
<td></td>
</tr>
<tr>
<td>SQLBulkOperations</td>
<td>ODBC</td>
<td>Performs bulk insertions and bulk bookmark operations, including update, delete, and fetch by bookmark.</td>
<td></td>
</tr>
<tr>
<td>SQLMoreResults</td>
<td>ODBC</td>
<td>Determines whether there are more result sets available and, if so, initializes processing for the next result set.</td>
<td></td>
</tr>
<tr>
<td>SQLGetDiagField</td>
<td>ISO 92</td>
<td>Returns additional diagnostic information (a single field of the diagnostic data structure).</td>
<td></td>
</tr>
<tr>
<td>SQLGetDiagRec</td>
<td>ISO 92</td>
<td>Returns additional diagnostic information (multiple fields of the diagnostic data structure).</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>SQLColumnPrivileges</td>
<td>ODBC</td>
<td>Returns a list of columns and associated privileges for one or more tables.</td>
<td></td>
</tr>
<tr>
<td>SQLColumns</td>
<td>Open Group</td>
<td>Returns the list of column names in specified tables.</td>
<td></td>
</tr>
<tr>
<td>SQLForeignKeys</td>
<td>ODBC</td>
<td>Returns a list of column names that make up foreign keys, if they exist for a specified table.</td>
<td></td>
</tr>
<tr>
<td>SQLPrimaryKeys</td>
<td>ODBC</td>
<td>Returns the list of column names that make up the primary key for a table.</td>
<td></td>
</tr>
<tr>
<td>SQLProcedureColumns</td>
<td>ODBC</td>
<td>Returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures.</td>
<td></td>
</tr>
<tr>
<td>SQLProcedures</td>
<td>ODBC</td>
<td>Returns the list of procedure names stored in a specific data source.</td>
<td></td>
</tr>
<tr>
<td>SQLSpecialColumns</td>
<td>Open Group</td>
<td>Returns information about the optimal set of columns that uniquely identifies a row in a specified table, or the columns that are automatically updated when any value in the row is updated by a transaction.</td>
<td></td>
</tr>
<tr>
<td>SQLStatistics</td>
<td>ISO 92</td>
<td>Returns statistics about a single table and the list of indexes associated with the table.</td>
<td></td>
</tr>
<tr>
<td>SQLTablePrivileges</td>
<td>ODBC</td>
<td>Returns a list of tables and the privileges associated with each table.</td>
<td></td>
</tr>
<tr>
<td>SQLTables</td>
<td>Open Group</td>
<td>Returns the list of table names stored in a specific data source.</td>
<td></td>
</tr>
<tr>
<td>SQLFreeStmt</td>
<td>ISO 92</td>
<td>Ends statement processing, discards pending results, and, optionally, frees all resources associated with the statement handle.</td>
<td></td>
</tr>
<tr>
<td>SQLCloseCursor</td>
<td>ISO 92</td>
<td>Closes a cursor that has been opened on a statement handle.</td>
<td></td>
</tr>
<tr>
<td>SQLCancel</td>
<td>ISO 92</td>
<td>Cancels the processing on a statement.</td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>Function name</td>
<td>Purpose</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Setting up data sources and</td>
<td>ConfigDriver</td>
<td>Installs or uninstalls a driver.</td>
<td></td>
</tr>
<tr>
<td>translators</td>
<td>ConfigDSN</td>
<td>Adds, modifies, or deletes a data source.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConfigTranslator</td>
<td>Returns a default translation option.</td>
<td></td>
</tr>
</tbody>
</table>

© 2016 Microsoft

## Setup DLL Function Summary

The following table describes setup DLL functions. For more information about the syntax and semantics for each function, see Setup DLL API Reference.

<table>
<thead>
<tr>
<th>Task</th>
<th>Function name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up data sources and</td>
<td>ConfigDriver</td>
<td>Installs or uninstalls a driver.</td>
</tr>
<tr>
<td>translators</td>
<td>ConfigDSN</td>
<td>Adds, modifies, or deletes a data source.</td>
</tr>
<tr>
<td></td>
<td>ConfigTranslator</td>
<td>Returns a default translation option.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

## Installer DLL Function Summary

The following table describes the functions in the installer DLL. For more information about the syntax and semantics for each function, see Installer DLL API Reference.

<table>
<thead>
<tr>
<th>Task</th>
<th>Function name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installing ODBC</td>
<td>SQLConfigDriver</td>
<td>Loads the driver-specific setup DLL.</td>
</tr>
<tr>
<td></td>
<td>SQLGetInstalledDrivers</td>
<td>Returns a list of installed drivers.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>SQLInstallDriverEx</td>
<td>Adds a driver to the system information.</td>
<td></td>
</tr>
<tr>
<td>SQLInstallDriverManager</td>
<td>Returns the target directory for the Driver Manager.</td>
<td></td>
</tr>
<tr>
<td>SQLInstallerError</td>
<td>Returns error or status information for the installer functions.</td>
<td></td>
</tr>
<tr>
<td>SQLInstallTranslatorEx</td>
<td>Adds a translator to the system information.</td>
<td></td>
</tr>
<tr>
<td>SQLPostInstallerError</td>
<td>Allows a driver or translator setup library to report errors.</td>
<td></td>
</tr>
<tr>
<td>SQLRemoveDriver</td>
<td>Removes a driver from the system information.</td>
<td></td>
</tr>
<tr>
<td>SQLRemoveDriverManager</td>
<td>Removes ODBC core components from the system information.</td>
<td></td>
</tr>
<tr>
<td>SQLRemoveTranslator</td>
<td>Removes the translator from the system information.</td>
<td></td>
</tr>
<tr>
<td>Configuring data sources</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLConfigDataSource</td>
<td>Calls the driver-specific setup DLL.</td>
<td></td>
</tr>
<tr>
<td>SQLCreateDataSource</td>
<td>Displays a dialog box to add a data source.</td>
<td></td>
</tr>
<tr>
<td>SQLGetConfigMode</td>
<td>Retrieves the configuration mode that indicates where the Odbc.ini entry listing DSN values is in the system information.</td>
<td></td>
</tr>
<tr>
<td>SQLGetPrivateProfileString</td>
<td>Writes a value to the system information.</td>
<td></td>
</tr>
<tr>
<td>SQLGetTranslator</td>
<td>Displays a dialog box to select a translator.</td>
<td></td>
</tr>
<tr>
<td>SQLManageDataSources</td>
<td>Displays a dialog box to configure data sources and drivers.</td>
<td></td>
</tr>
<tr>
<td>SQLReadFileDSN</td>
<td>Reads information from file DSNs.</td>
<td></td>
</tr>
<tr>
<td>SQLRemoveDefaultDataSource</td>
<td>Removes the default data source.</td>
<td></td>
</tr>
<tr>
<td>SQLRemoveDSNFromIni</td>
<td>Removes a data source.</td>
<td></td>
</tr>
<tr>
<td>SQLSetConfigMode</td>
<td>Sets the configuration mode that...</td>
<td></td>
</tr>
</tbody>
</table>
The following table describes ODBC Service Provider interface functions. For more information about the syntax and semantics for each function, see ODBC Service Provider Interface (SPI) Reference.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLSetConnectAttrForDbcInfo</td>
<td>Same as SQLSetConnectAttr, but it sets the attribute on the connection information token instead of on the connection handle.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SQLSetDriverConnectInfo</td>
<td>Sets the connection string into the connection info token for an application’s SQLDriverConnect call.</td>
</tr>
<tr>
<td>SQLSetConnectInfo</td>
<td>Sets the data source, user ID, and password into the connection info token for an application’s SQLConnect call.</td>
</tr>
<tr>
<td>SQLGetPoolID</td>
<td>Retrieves the pool ID.</td>
</tr>
<tr>
<td>SQLRateConnection</td>
<td>Determines if a driver can reuse an existing connection in the connection pool.</td>
</tr>
<tr>
<td>SQLPoolConnect</td>
<td>Create a new connection if no connection in the pool can be reused.</td>
</tr>
<tr>
<td>SQLCleanupConnectionPoolID</td>
<td>Informs a driver that a pool ID was timed out.</td>
</tr>
</tbody>
</table>

The topics in this section describe each ODBC function in alphabetical order. Each function is defined as a C programming language function. Descriptions include the following:

- Purpose
- ODBC version
- Standard CLI conformance level
- Syntax
- Arguments
- Return values
- Diagnostics
- Comments about usage and implementation
- Code example
- References to related functions

The standard CLI conformance level can be one of the following: ISO 92, Open Group, ODBC, or Deprecated. A function tagged as ISO 92–conformant also appears in Open Group version 1, because Open Group is a pure superset of ISO 92. A function tagged as Open Group-compliant also appears in ODBC 3.x, because ODBC 3.x is a pure superset of Open Group version 1. A function tagged as ODBC-compliant appears in neither standard. A function tagged as deprecated has been deprecated in ODBC 3.x.

Handling of diagnostic information is described in the SQLGetDiagField function description. The text associated with SQLSTATE values is included to provide a description of the condition but is not intended to prescribe specific text.
Note

For driver-specific information about ODBC functions, see the section for the driver.

This section contains topics for the following functions:

- SQLAllocConnect Function
- SQLAllocEnv Function
- SQLAllocHandle Function
- SQLAllocStmt Function
- SQLBindCol Function
- SQLBindParameter Function
- SQLBrowseConnect Function
- SQLBulkOperations Function
- SQLCancel Function
- SQLCancelHandle Function
- SQLCloseCursor Function
- SQLColAttribute Function
- SQLColAttributes Function
- SQLColumnPrivileges Function
- SQLColumns Function
- SQLCompleteAsync Function
- SQLConnect Function
- SQLCopyDesc Function
- SQLDataSources Function
- SQLDescribeCol Function
- SQLDescribeParam Function
- SQLEndTran Function
- SQLDriverConnect Function
- SQLDrivers Function
- SQLError Function
- SQLExecDirect Function
- SQLExecute Function
- SQLExtendedFetch Function
- SQLFetch Function
- SQLFetchScroll Function
- SQLForeignKeys Function
- SQLFreeConnect Function
- SQLFreeEnv Function
- SQLFreeHandle Function
- SQLFreeStmt Function
- SQLGetConnectAttr Function
- SQLGetConnectOption Function
- SQLGetCursorName Function
- SQLGetData Function
- SQLGetDescField Function
- SQLGetDescRec Function
- SQLGetDiagField Function
- SQLGetDiagRec Function
- SQLGetEnvAttr Function
- SQLGetFunctions Function
- SQLGetInfo Function
- SQLGetStmtAttr Function
- SQLGetStmtOption Function
- SQLGetTypeInfo Function
- SQLMoreResults Function
- SQLNativeSql Function
- SQLNumParams Function
- SQLNumResultCols Function
- SQLParamData Function
- SQLParamOptions Function
- SQLPrepare Function
SQLAllocConnect Function

**Conformance**
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

**Summary**
In ODBC 3.x, the ODBC 2.x function SQLAllocConnect has been replaced by SQLAllocHandle. For more information, see SQLAllocHandle Function.

*Note*
For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLAllocEnv Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

Summary
In ODBC 3.x, the ODBC 2.x function SQLAllocEnv has been replaced by SQLAllocHandle. For more information, see SQLAllocHandle Function.

Note
For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLAllocHandle Function

Conformance
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary
SQLAllocHandle allocates an environment, connection, statement, or descriptor handle.

Note
This function is a generic function for allocating handles that replaces the ODBC 2.0 functions
To allow applications calling `SQLAllocHandle` to work with ODBC 2.x drivers, a call to `SQLAllocHandle` is mapped in the Driver Manager to `SQLAllocConnect`, `SQLAllocEnv`, or `SQLAllocStmt`, as appropriate. For more information, see "Comments." For more information about what the Driver Manager maps this function to when an ODBC 3.x application is working with an ODBC 2.x driver, see Mapping Replacement Functions for Backward Compatibility of Applications.

**Syntax**

```c
SQLRETURN SQLAllocHandle(
    SQLSMALLINT HandleType,
    SQLHANDLE InputHandle,
    SQLHANDLE * OutputHandlePtr);
```

**Arguments**

*HandleType*

[Input] The type of handle to be allocated by `SQLAllocHandle`. Must be one of the following values:

- SQL_HANDLE_DBC
- SQL_HANDLE_DBC_INFO_TOKEN
- SQL_HANDLE_DESC
- SQL_HANDLE_ENV
- SQL_HANDLE_STMT

*InputHandle*

[Input] The input handle in whose context the new handle is to be allocated. If `HandleType` is SQL_HANDLE_ENV, this is SQL_NULL_HANDLE. If `HandleType` is SQL_HANDLE_DBC, this must be an environment handle, and if it is SQL_HANDLE_STMT or SQL_HANDLE_DESC, it must be a connection handle.

*OutputHandlePtr*

[Output] Pointer to a buffer in which to return the handle to the newly allocated data structure.

**Returns**

`SQL_SUCCESS`, `SQL_SUCCESS_WITH_INFO`, `SQL_INVALID_HANDLE`, or `SQL_ERROR`.
When allocating a handle other than an environment handle, if `SQLAllocHandle` returns SQL_ERROR, it sets `OutputHandlePtr` to SQL_NULL_HDBC, SQL_NULL_HSTMT, or SQL_NULL_HDESC, depending on the value of `HandleType`, unless the output argument is a null pointer. The application can then obtain additional information from the diagnostic data structure associated with the handle in the `InputHandle` argument.

Environment Handle Allocation Errors

Environment allocation occurs both within the Driver Manager and within each driver. The error returned by `SQLAllocHandle` with a `HandleType` of SQL_HANDLE_ENV depends on the level in which the error occurred.

If the Driver Manager cannot allocate memory for `*OutputHandlePtr` when `SQLAllocHandle` with a `HandleType` of SQL_HANDLE_ENV is called, or the application provides a null pointer for `OutputHandlePtr`, `SQLAllocHandle` returns SQL_ERROR. The Driver Manager sets `*OutputHandlePtr` to SQL_NULL_HENV (unless the application provided a null pointer, which returns SQL_ERROR). There is no handle with which to associate additional diagnostic information.

The Driver Manager does not call the driver-level environment handle allocation function until the application calls `SQLConnect`, `SQLBrowseConnect`, or `SQLDriverConnect`. If an error occurs in the driver-level `SQLAllocHandle` function, then the Driver Manager–level `SQLConnect`, `SQLBrowseConnect`, or `SQLDriverConnect` function returns SQL_ERROR. The diagnostic data structure contains SQLSTATE IM004 (Driver’s `SQLAllocHandle` failed). The error is returned on a connection handle.

For more information about the flow of function calls between the Driver Manager and a driver, see `SQLConnect Function`.

Diagnostics

When `SQLAllocHandle` returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling `SQLGetDiagRec` with the appropriate `HandleType` and `Handle` set to the value of `InputHandle`. SQL_SUCCESS_WITH_INFO (but not SQL_ERROR) can be returned for the `OutputHandle` argument. The following table lists the SQLSTATE values typically returned by `SQLAllocHandle` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
<td>(DM) The <code>HandleType</code> argument was SQL_HANDLE_STMT or SQL_HANDLE_DESC, but the connection specified by the <code>InputHandle</code> argument was not open. The connection process must be completed successfully (and the connection must be open) for the driver to allocate a statement or descriptor handle.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the <em>MessageText</em> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>(DM) The Driver Manager was unable to allocate memory for the specified handle. The driver was unable to allocate memory for the specified handle.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>(DM) The OutputHandlePtr argument was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) The HandleType argument was SQL_HANDLE_DBC, and SQLSetEnvAttr has not been called to set the SQL_ODBC_VERSION environment attribute. (DM) An asynchronously executing function was called for the InputHandle and was still executing when the SQLAllocHandle function was called with HandleType set to SQL_HANDLE_STMT or SQL_HANDLE_DESC.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The HandleType argument was SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC; and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY014</td>
<td>Limit on the number of handles exceeded</td>
<td>The driver-defined limit for the number of handles that can be allocated for the type of handle indicated by the HandleType argument has been reached.</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute/options identifier</td>
<td>(DM) The HandleType argument was not: SQL_HANDLE_ENV, SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The HandleType argument was SQL_HANDLE_DESC and the driver was an ODBC 2.x driver.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not</td>
<td>(DM) The HandleType argument was SQL_HANDLE_STMT,</td>
</tr>
</tbody>
</table>
support this function and the driver was not a valid ODBC driver.

(DM) The HandleType argument was SQL_HANDLE_DESC, and the driver does not support allocating a descriptor handle.

Comments

**SQLAllocHandle** is used to allocate handles for environments, connections, statements, and descriptors, as described in the following sections. For general information about handles, see Handles.

More than one environment, connection, or statement handle can be allocated by an application at a time if multiple allocations are supported by the driver. In ODBC, no limit is defined on the number of environment, connection, statement, or descriptor handles that can be allocated at any one time. Drivers may impose a limit on the number of a certain type of handle that can be allocated at a time; for more information, see the driver documentation.

If the application calls **SQLAllocHandle** with *OutputHandlePtr set to an environment, connection, statement, or descriptor handle that already exists, the driver overwrites the information associated with the handle, unless the application is using connection pooling (see "Allocating an Environment Attribute for Connection Pooling" later in this section). The Driver Manager does not check to see whether the handle entered in *OutputHandlePtr is already being used, nor does it check the previous contents of a handle before overwriting them.

**Note**

It is incorrect ODBC application programming to call **SQLAllocHandle** two times with the same application variable defined for *OutputHandlePtr without calling **SQLFreeHandle** to free the handle before reallocating it. Overwriting ODBC handles in such a manner could lead to inconsistent behavior or errors on the part of ODBC drivers.

On operating systems that support multiple threads, applications can use the same environment, connection, statement, or descriptor handle on different threads. Drivers must therefore support safe, multithread access to this information; one way to achieve this, for example, is by using a critical section or a semaphore. For more information about threading, see Multithreading.

**SQLAllocHandle** does not set the SQL_ATTR_ODBC_VERSION environment attribute when it is called to allocate an environment handle; the environment attribute must be set by the application, or SQLSTATE HY010 (Function sequence error) will be returned when **SQLAllocHandle** is called to allocate a connection handle.

For standards-compliant applications, **SQLAllocHandle** is mapped to **SQLAllocHandleStd** at compile time. The difference between these two functions is that **SQLAllocHandleStd** sets the SQL_ATTR_ODBC_VERSION environment attribute to SQL_OV_ODBC3 when it is called with the HandleType argument set to SQL_HANDLE_ENV. This is done because standards-compliant applications are always ODBC 3.x applications. Moreover, the standards do not require the application version to be registered. This is the only difference between these two functions; otherwise, they are identical. **SQLAllocHandleStd** is mapped to **SQLAllocHandle** inside the driver manager. Therefore, third-party drivers do not have to implement **SQLAllocHandleStd**.

ODBC 3.8 applications should use:
- **SQLAllocHandle** and not **SQLAllocHandleStd** to allocate an environment handle.
- **SQLSetEnvAttr** to set the SQL_ATTR_ODBC_VERSION environment attribute to SQL_OV_ODBC3_80.

### Allocating an Environment Handle

An environment handle provides access to global information such as valid connection handles and active connection handles. For general information about environment handles, see [Environment Handles](#).

To request an environment handle, an application calls **SQLAllocHandle** with a **HandleType** of SQL_HANDLE_ENV and an **InputHandle** of SQL_NULL_HANDLE. The driver allocates memory for the environment information and passes the value of the associated handle back in the *OutputHandlePtr* argument. The application passes the *OutputHandle* value in all subsequent calls that require an environment handle argument. For more information, see [Allocating the Environment Handle](#).

Under a Driver Manager's environment handle, if there already exists a driver's environment handle, then **SQLAllocHandle** with a **HandleType** of SQL_HANDLE_ENV is not called in that driver when a connection is made, only **SQLAllocHandle** with a **HandleType** of SQL_HANDLE_DBC. If a driver's environment handle does not exist under the Driver Manager's environment handle, both **SQLAllocHandle** with a **HandleType** of SQL_HANDLE_ENV and **SQLAllocHandle** with a **HandleType** of SQL_HANDLE_DBC are called in the driver when the first connection handle of the environment is connected to the driver.

When the Driver Manager processes the **SQLAllocHandle** function with a **HandleType** of SQL_HANDLE_ENV, it checks the **Trace** keyword in the [ODBC] section of the system information. If it is set to 1, the Driver Manager enables tracing for the current application. If the trace flag is set, tracing starts when the first environment handle is allocated and ends when the last environment handle is freed. For more information, see [Configuring Data Sources](#).

After allocating an environment handle, an application must call **SQLSetEnvAttr** on the environment handle to set the SQL_ATTR_ODBC_VERSION environment attribute. If this attribute is not set before **SQLAllocHandle** is called to allocate a connection handle on the environment, the call to allocate the connection will return SQLSTATE HY010 (Function sequence error). For more information, see [Declaring the Application's ODBC Version](#).

### Allocating Shared Environments for Connection Pooling

Environments can be shared among multiple components on a single process. A shared environment can be used by more than one component at the same time. When a component uses a shared environment, it can use pooled connections, which allow it to allocate and use an existing connection without re-creating that connection.

Before allocating a shared environment that can be used for connection pooling, an application must call **SQLSetEnvAttr** to set the SQL_ATTR_CONNECTION_POOLING environment attribute to SQL_CP_ONE_PER_DRIVER or SQL_CP_ONE_PER_HENV. **SQLSetEnvAttr** in this case is called with **EnvironmentHandle** set to null, which makes the attribute a process-level attribute.

After connection pooling has been enabled, an application calls **SQLAllocHandle** with the **HandleType** argument set to SQL_HANDLE_ENV. The environment allocated by this call will be an implicit shared environment because connection pooling has been enabled.
When a shared environment is allocated, the environment that will be used is not determined until `SQLAllocHandle` with a `HandleType` of `SQL_HANDLE_DBC` is called. At that point, the Driver Manager tries to find an existing environment that matches the environment attributes requested by the application. If no such environment exists, one is created as a shared environment. The Driver Manager maintains a reference count for each shared environment; the count is set to 1 when the environment is first created. If a matching environment is found, the handle of that environment is returned to the application and the reference count is incremented. An environment handle allocated in this manner can be used in any ODBC function that accepts an environment handle as an input argument.

**Allocating a Connection Handle**

A connection handle provides access to information such as the valid statement and descriptor handles on the connection and whether a transaction is currently open. For general information about connection handles, see [Connection Handles](#).

To request a connection handle, an application calls `SQLAllocHandle` with a `HandleType` of `SQL_HANDLE_DBC`. The `InputHandle` argument is set to the environment handle that was returned by the call to `SQLAllocHandle` that allocated that handle. The driver allocates memory for the connection information and passes the value of the associated handle back in `*OutputHandlePtr`. The application passes the `*OutputHandlePtr` value in all subsequent calls that require a connection handle. For more information, see [Allocating a Connection Handle](#).

The Driver Manager processes the `SQLAllocHandle` function and calls the driver's `SQLAllocHandle` function when the application calls `SQLConnect`, `SQLBrowseConnect`, or `SQLDriverConnect`. (For more information, see [SQLConnect Function](#).)

If the SQL_ATTR_ODBC_VERSION environment attribute is not set before `SQLAllocHandle` is called to allocate a connection handle on the environment, the call to allocate the connection will return SQLSTATE HY010 (Function sequence error).

When an application calls `SQLAllocHandle` with the `InputHandle` argument set to `SQL_HANDLE_DBC` and also set to a shared environment handle, the Driver Manager tries to find an existing shared environment that matches the environment attributes set by the application. If no such environment exists, one is created, with a reference count (maintained by the Driver Manager) of 1. If a matching shared environment is found, that handle is returned to the application and its reference count is incremented.

The actual connection that will be used is not determined by the Driver Manager until `SQLConnect` or `SQLDriverConnect` is called. The Driver Manager uses the connection options in the call to `SQLConnect` (or the connection keywords in the call to `SQLDriverConnect`) and the connection attributes set after connection allocation to determine which connection in the pool should be used. For more information, see [SQLConnect Function](#).

**Allocating a Statement Handle**

A statement handle provides access to statement information, such as error messages, the cursor name, and status information for SQL statement processing. For general information about statement handles, see [Statement Handles](#).

To request a statement handle, an application connects to a data source and then calls `SQLAllocHandle` before it submits SQL statements. In this call, `HandleType` should be set to
SQL_HANDLE_STMT and *InputHandle* should be set to the connection handle that was returned by the call to *SQLAllocHandle* that allocated that handle. The driver allocates memory for the statement information, associates the statement handle with the specified connection, and passes the value of the associated handle back in *OutputHandlePtr*. The application passes the *OutputHandlePtr* value in all subsequent calls that require a statement handle. For more information, see Allocating a Statement Handle.

When the statement handle is allocated, the driver automatically allocates a set of four descriptors and assigns the handles for these descriptors to the SQL_ATTR_APP_ROW_DESC, SQL_ATTR_APP_PARAM_DESC, SQL_ATTR_IMP_ROW_DESC, and SQL_ATTR_IMP_PARAM_DESC statement attributes. These are referred to as implicitly allocated descriptors. To allocate an application descriptor explicitly, see the following section, "Allocating a Descriptor Handle."

### Allocating a Descriptor Handle

When an application calls *SQLAllocHandle* with a *HandleType* of SQL_HANDLE_DESC, the driver allocates an application descriptor. These are referred to as explicitly allocated descriptors. The application directs a driver to use an explicitly allocated application descriptor instead of an automatically allocated one for a given statement handle by calling the *SQLSetStmtAttr* function with the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC attribute. An implementation descriptor cannot be allocated explicitly, nor can an implementation descriptor be specified in an *SQLSetStmtAttr* function call.

Explicitly allocated descriptors are associated with a connection handle instead of a statement handle (as automatically allocated descriptors are). Descriptors remain allocated only when an application is actually connected to the database. Because explicitly allocated descriptors are associated with a connection handle, an application can associate an explicitly allocated descriptor with more than one statement within a connection. An implicitly allocated application descriptor, on the other hand, cannot be associated with more than one statement handle. (It cannot be associated with any statement handle other than the one that it was allocated for.) Explicitly allocated descriptor handles can be freed explicitly either by the application or by calling *SQLFreeHandle* with a *HandleType* of SQL_HANDLE_DESC, or implicitly when the connection is closed.

When the explicitly allocated descriptor is freed, the implicitly allocated descriptor is again associated with the statement. (The SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC attribute for that statement is again set to the implicitly allocated descriptor handle.) This is true for all statements that were associated with the explicitly allocated descriptor on the connection.

For more information about descriptors, see Descriptors.

### Code Example

See Sample ODBC Program, SQLBrowseConnect Function, SQLConnect Function, and SQLSetCursorName Function.

### Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Executing an SQL statement | SQLExecDirect Function
Executing a prepared SQL statement | SQLExecute Function
Freeing an environment, connection, statement, or descriptor handle | SQLFreeHandle Function
Preparing a statement for execution | SQLPrepare Function
Setting a connection attribute | SQLSetConnectAttr Function
Setting a descriptor field | SQLSetDescField Function
Setting an environment attribute | SQLSetEnvAttr Function
Setting a statement attribute | SQLSetStmtAttr Function

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLAllocStmt Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

Summary
In ODBC 3.x, the ODBC 2.x function SQLAllocStmt has been replaced by SQLAllocHandle. For more information, see SQLAllocHandle Function.

Note
For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft
## SQLBindCol Function

### Conformance

**Version Introduced:** ODBC 1.0  
**Standards Compliance:** ISO 92

### Summary

**SQLBindCol** binds application data buffers to columns in the result set.

### Syntax

```c
SQLRETURN SQLBindCol(
    SQLHSTMT StatementHandle,
    SQLUSMALLINT ColumnNumber,
    SQLSMALLINT TargetType,
    SQLPOINTER TargetValuePtr,
    SQLLEN BufferLength,
    SQLLEN * StrLen_or_Ind);
```

### Arguments

**StatementHandle**  
*Input* Statement handle.

**ColumnNumber**  
*Input* Number of the result set column to bind. Columns are numbered in increasing column order starting at 0, where column 0 is the bookmark column. If bookmarks are not used — that is, the SQL_ATTR_USE_BOOKMARKS statement attribute is set to SQL_UB_OFF — then column numbers start at 1.

**TargetType**  
*Input* The identifier of the C data type of the *TargetValuePtr* buffer. When it is retrieving data from the data source with **SQLFetch**, **SQLFetchScroll**, **SQLBulkOperations**, or **SQLSetPos**, the driver converts the data to this type; when it sends data to the data source with **SQLBulkOperations** or **SQLSetPos**, the driver converts the data from this type. For a list of valid C data types and type identifiers, see the C Data Types section in Appendix D: Data Types.

If the **TargetType** argument is an interval data type, the default interval leading precision (2) and the default interval seconds precision (6), as set in the SQL_DESC_DATETIME_INTERVAL_PRECISION and SQL_DESC_PRECISION fields of the ARD, respectively, are used for the data. If the **TargetType** argument is SQL_C_NUMERIC, the default precision (driver-defined) and default scale (0), as set in the SQL_DESCPRECISION and SQL_DESC_SCALE fields of the ARD, are used for the data. If any default precision or scale is not appropriate, the application should explicitly set the appropriate descriptor field by a call to **SQLSetDescField** or **SQLSetDescRec**.

You can also specify an extended C data type. For more information, see C Data Types in ODBC.
**TargetValuePtr**

[Deferred Input/Output] Pointer to the data buffer to bind to the column. SQLFetch and SQLFetchScroll return data in this buffer. SQLBulkOperations returns data in this buffer when Operation is SQL_FETCH_BY_BOOKMARK; it retrieves data from this buffer when Operation is SQL_ADD or SQL_UPDATE_BY_BOOKMARK. SQLSetPos returns data in this buffer when Operation is SQL_REFRESH; it retrieves data from this buffer when Operation is SQL_UPDATE.

If TargetValuePtr is a null pointer, the driver unbinds the data buffer for the column. An application can unbind all columns by calling SQLFreeStmt with the SQL_UNBIND option. An application can unbind the data buffer for a column but still have a length/indicator buffer bound for the column, if the TargetValuePtr argument in the call to SQLBindCol is a null pointer but the StrLen_or_IndPtr argument is a valid value.

**BufferLength**

[Input] Length of the *TargetValuePtr* buffer in bytes.

The driver uses BufferLength to avoid writing past the end of the *TargetValuePtr* buffer when it returns variable-length data, such as character or binary data. Notice that the driver counts the null-termination character when it returns character data to *TargetValuePtr*. *TargetValuePtr* must therefore contain space for the null-termination character or the driver will truncate the data.

When the driver returns fixed-length data, such as an integer or a date structure, the driver ignores BufferLength and assumes the buffer is large enough to hold the data. Therefore, it is important for the application to allocate a large enough buffer for fixed-length data or the driver will write past the end of the buffer.

SQLBindCol returns SQLSTATE HY090 (Invalid string or buffer length) when BufferLength is less than 0 but not when BufferLength is 0. However, if TargetType specifies a character type, an application should not set BufferLength to 0, because ISO CLI–compliant drivers return SQLSTATE HY090 (Invalid string or buffer length) in that case.

**StrLen_or_IndPtr**

[Deferred Input/Output] Pointer to the length/indicator buffer to bind to the column. SQLFetch and SQLFetchScroll return a value in this buffer. SQLBulkOperations retrieves a value from this buffer when Operation is SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or SQL_DELETE_BY_BOOKMARK. SQLBulkOperations returns a value in this buffer when Operation is SQL_FETCH_BY_BOOKMARK. SQLSetPos returns a value in this buffer when Operation is SQL_REFRESH; it retrieves a value from this buffer when Operation is SQL_UPDATE.

SQLFetch, SQLFetchScroll, SQLBulkOperations, and SQLSetPos can return the following values in the length/indicator buffer:

- The length of the data available to return
- SQL_NO_TOTAL
- SQL_NULL_DATA

The application can put the following values in the length/indicator buffer for use with SQLBulkOperations or SQLSetPos:

- The length of the data being sent
- SQL_NTS
- SQL_NULL_DATA
- SQL_DATA_AT_EXEC
- The result of the SQL_LEN_DATA_AT_EXEC macro
- SQL_COLUMN_IGNORE

If the indicator buffer and the length buffer are separate buffers, the indicator buffer can return only SQL_NULL_DATA, whereas the length buffer can return all other values.

For more information, see SQLBulkOperations Function, SQLFetch Function, SQLSetPos Function, and Using Length/Indicator Values.

If StrLen_or_IndPtr is a null pointer, no length or indicator value is used. This is an error when fetching data and the data is NULL.

See ODBC 64-Bit Information, if your application will run on a 64-bit operating system.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When SQLBindCol returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values typically returned by SQLBindCol and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>(DM) The ColumnNumber argument was 0, and the TargetType argument was not SQL_C_BOOKMARK or SQL_C_VARBOOKMARK.</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>The value specified for the argument ColumnNumber exceeded the maximum number of columns in the result set.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific</td>
</tr>
</tbody>
</table>
SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY003</td>
<td>Invalid application buffer type</td>
<td>The argument TargetType was neither a valid data type nor SQL_C_DEFAULT.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when SQLBindCol was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) An asynchronously executing function was called for the StatementHandle and was still executing when this function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value specified for the argument BufferLength was less than 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The driver was an ODBC 2.x driver, the ColumnNumber argument was set to 0, and the value specified for the argument BufferLength was not equal to 4.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The driver or data source does not support the conversion specified by the combination of the TargetType argument and the driver-specific SQL data type of the corresponding column.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The argument ColumnNumber was 0 and the driver does</td>
</tr>
</tbody>
</table>
The driver supports only ODBC 2.x and the argument TargetType was one of the following:

SQL_C_NUMERIC SQL_C_SBIGINT SQL_C_UBIGINT

and any of the interval C data types listed in C Data Types in Appendix D: Data Types.

The driver only supports ODBC versions prior to 3.50, and the argument TargetType was SQL_C_GUID.

**HYT01**  Connection timeout expired  The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.

**IM001**  Driver does not support this function  (DM) The driver associated with the StatementHandle does not support the function.

## Comments

**SQLBindCol** is used to associate, or *bind*, columns in the result set to data buffers and length/indicator buffers in the application. When the application calls SQLFetch, SQLFetchScroll, or SQLSetPos to fetch data, the driver returns the data for the bound columns in the specified buffers; for more information, see SQLFetch Function. When the application calls SQLBulkOperations to update or insert a row or SQLSetPos to update a row, the driver retrieves the data for the bound columns from the specified buffers; for more information, see SQLBulkOperations Function or SQLSetPos Function. For more information about binding, see Retrieving Results (Basic).

Notice that columns do not have to be bound to retrieve data from them. An application can also call SQLGetData to retrieve data from columns. Although it is possible to bind some columns in a row and call SQLGetData for others, this is subject to some restrictions. For more information, see SQLGetData.

## Binding, Unbinding, and Rebinding Columns

A column can be bound, unbound, or rebound at any time, even after data has been fetched from the result set. The new binding takes effect the next time that a function that uses bindings is called. For example, suppose an application binds the columns in a result set and calls SQLFetch. The driver returns the data in the bound buffers. Now suppose the application binds the columns to a different set of buffers. The driver does not put the data for the just-fetched row in the newly bound buffers. Instead, it waits until SQLFetch is called again and then places the data for the next row in the newly bound buffers.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
</table>

The statement attribute SQL_ATTR_USE_BOOKMARKS should always be set before binding a column to column 0. This is not required but is strongly recommended.
Binding Columns

To bind a column, an application calls SQLBindCol and passes the column number, type, address, and length of a data buffer, and the address of a length/indicator buffer. For information about how these addresses are used, see "Buffer Addresses," later in this section. For more information about binding columns, see Using SQLBindCol.

The use of these buffers is deferred; that is, the application binds them in SQLBindCol but the driver accesses them from other functions — namely SQLBulkOperations, SQLFetch, SQLFetchScroll, or SQLSetPos. It is the application's responsibility to make sure that the pointers specified in SQLBindCol remain valid as long as the binding remains in effect. If the application allows these pointers to become invalid — for example, it frees a buffer — and then calls a function that expects them to be valid, the consequences are undefined. For more information, see Deferred Buffers.

The binding remains in effect until it is replaced by a new binding, the column is unbound, or the statement is freed.

Unbinding Columns

To unbind a single column, an application calls SQLBindCol with ColumnNumber set to the number of that column and TargetValuePtr set to a null pointer. If ColumnNumber refers to an unbound column, SQLBindCol still returns SQL_SUCCESS.

To unbind all columns, an application calls SQLFreeStmt with fOption set to SQL_UNBIND. This can also be accomplished by setting the SQL_DESC_COUNT field of the ARD to zero.

Rebinding Columns

An application can perform either of two operations to change a binding:

- Call SQLBindCol to specify a new binding for a column that is already bound. The driver overwrites the old binding with the new one.
- Specify an offset to be added to the buffer address that was specified by the binding call to SQLBindCol. For more information, see the next section, "Binding Offsets."

Binding Offsets

A binding offset is a value that is added to the addresses of the data and length/indicator buffers (as specified in the TargetValuePtr and StrLen_or_IndPtr argument) before they are dereferenced. When offsets are used, the bindings are a "template" of how the application's buffers are laid out, and the application can move this "template" to different areas of memory by changing the offset. Because the same offset is added to each address in each binding, the relative offsets between buffers for different columns must be the same within each set of buffers. This is always true when row-wise binding is used; the application must carefully lay out its buffers for this to be true when column-wise binding is used.
Using a binding offset has basically the same effect as rebinding a column by calling `SQLBindCol`. The difference is that a new call to `SQLBindCol` specifies new addresses for the data buffer and length/indicator buffer, whereas use of a binding offset does not change the addresses but just adds an offset to them. The application can specify a new offset whenever it wants, and this offset is always added to the originally bound addresses. In particular, if the offset is set to 0 or if the statement attribute is set to a null pointer, the driver uses the originally bound addresses.

To specify a binding offset, the application sets the `SQL_ATTR_ROW_BIND_OFFSET_PTR` statement attribute to the address of an `SQLINTEGER` buffer. Before the application calls a function that uses bindings, it puts an offset in bytes in this buffer. To determine the address of the buffer to use, the driver adds the offset to the address in the binding. The sum of the address and the offset must be a valid address, but the address to which the offset is added does not have to be valid. For more information about how binding offsets are used, see "Buffer Addresses," later in this section.

### Binding Arrays

If the rowset size (the value of the `SQL_ATTR_ROW_ARRAY_SIZE` statement attribute) is greater than 1, the application binds arrays of buffers instead of single buffers. For more information, see Block Cursors.

The application can bind arrays in two ways:

- Bind an array to each column. This is referred to as column-wise binding because each data structure (array) contains data for a single column.

- Define a structure to hold the data for a whole row and bind an array of these structures. This is referred to as row-wise binding because each data structure contains the data for a single row.

Each array of buffers must have at least as many elements as the size of the rowset.

---

**Note**

An application must verify that alignment is valid. For more information about alignment considerations, see Alignment.

### Column-Wise Binding

In column-wise binding, the application binds separate data and length/indicator arrays to each column.

To use column-wise binding, the application first sets the `SQL_ATTR_ROW_BIND_TYPE` statement attribute to `SQL_BIND_BY_COLUMN`. (This is the default.) For each column to be bound, the application performs the following steps:

1. Allocates a data buffer array.

2. Allocates an array of length/indicator buffers.
If the application writes directly to descriptors when column-wise binding is used, separate arrays can be used for length and indicator data.

3. Calls SQLBindCol with the following arguments:
   - \textit{TargetType} is the type of a single element in the data buffer array.
   - \textit{TargetValuePtr} is the address of the data buffer array.
   - \textit{BufferLength} is the size of a single element in the data buffer array. The \textit{BufferLength} argument is ignored when the data is fixed-length data.
   - \textit{StrLen_or_IndPtr} is the address of the length/indicator array.

For more information about how this information is used, see "Buffer Addresses," later in this section. For more information about column-wise binding, see Column-Wise Binding.

Row–Wise Binding

In row-wise binding, the application defines a structure that contains data and length/indicator buffers for each column to be bound.

To use row-wise binding, the application performs the following steps:

1. Defines a structure to hold a single row of data (including both data and length/indicator buffers) and allocates an array of these structures.

   \textbf{Note}
   
   If the application writes directly to descriptors when row-wise binding is used, separate fields can be used for length and indicator data.

2. Sets the SQL\_ATTR\_ROW\_BIND\_TYPE statement attribute to the size of the structure that contains a single row of data or to the size of an instance of a buffer into which the results columns will be bound. The length must include space for all the bound columns, and any padding of the structure or buffer, to make sure that when the address of a bound column is incremented with the specified length, the result will point to the beginning of the same column in the next row. When using the sizeof operator in ANSI C, this behavior is guaranteed.

3. Calls SQLBindCol with the following arguments for each column to be bound:
   - \textit{TargetType} is the type of the data buffer member to be bound to the column.
   - \textit{TargetValuePtr} is the address of the data buffer member in the first array element.
   - \textit{BufferLength} is the size of the data buffer member.
   - \textit{StrLen_or_IndPtr} is the address of the length/indicator member to be bound.

For more information about how this information is used, see "Buffer Addresses," later in this section. For more information about column-wise binding, see Row–Wise Binding.
Buffer Addresses

The buffer address is the actual address of the data or length/indicator buffer. The driver calculates the buffer address just before it writes to the buffers (such as during fetch time). It is calculated from the following formula, which uses the addresses specified in the `TargetValuePtr` and `StrLen_or_IndPtr` arguments, the binding offset, and the row number:

$$\text{Bound Address} + \text{Binding Offset} + ((\text{Row Number} - 1) \times \text{Element Size})$$

where the formula's variables are defined as described in the following table.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bound Address</td>
<td>For data buffers, the address specified with the <code>TargetValuePtr</code> argument in <code>SQLBindCol</code>.</td>
</tr>
<tr>
<td></td>
<td>For length/indicator buffers, the address specified with the <code>StrLen_or_IndPtr</code> argument in <code>SQLBindCol</code>. For more information, see &quot;Additional Comments&quot; in the &quot;Descriptors and SQLBindCol&quot; section.</td>
</tr>
<tr>
<td></td>
<td>If the bound address is 0, no data value is returned, even if the address as calculated by the previous formula is nonzero.</td>
</tr>
<tr>
<td>Binding Offset</td>
<td>If row-wise binding is used, the value stored at the address specified with the <code>SQL_ATTR_ROW_BIND_OFFSET_PTR</code> statement attribute.</td>
</tr>
<tr>
<td></td>
<td>If column-wise binding is used or if the value of the <code>SQL_ATTR_ROW_BIND_OFFSET_PTR</code> statement attribute is a null pointer, <code>Binding Offset</code> is 0.</td>
</tr>
<tr>
<td>Row Number</td>
<td>The 1-based number of the row in the rowset. For single-row fetches, which are the default, this is 1.</td>
</tr>
<tr>
<td>Element Size</td>
<td>The size of an element in the bound array.</td>
</tr>
<tr>
<td></td>
<td>If column-wise binding is used, this is <code>sizeof(SQLINTEGER)</code> for length/indicator buffers. For data buffers, it is the value of the <code>BufferLength</code> argument in <code>SQLBindCol</code> if the data type is variable length, and the size of the data type if the data type is fixed length.</td>
</tr>
<tr>
<td></td>
<td>If row-wise binding is used, this is the value of the <code>SQL_ATTR_ROW_BIND_TYPE</code> statement attribute for both data and length/indicator buffers.</td>
</tr>
</tbody>
</table>

Descriptors and SQLBindCol

The following sections describe how `SQLBindCol` interacts with descriptors.

⚠️ Caution

Calling `SQLBindCol` for one statement can affect other statements. This occurs when the ARD
associated with the statement is explicitly allocated and is also associated with other statements. Because SQLBindCol modifies the descriptor, the modifications apply to all statements with which this descriptor is associated. If this is not the required behavior, the application should dissociate this descriptor from the other statements before it calls SQLBindCol.

Argument Mappings

Conceptually, SQLBindCol performs the following steps in sequence:

1. Calls SQLGetStmtAttr to obtain the ARD handle.

2. Calls SQLGetDescField to get this descriptor's SQL_DESC_COUNT field, and if the value in the ColumnNumber argument exceeds the value of SQL_DESC_COUNT, calls SQLSetDescField to increase the value of SQL_DESC_COUNT to ColumnNumber.

3. Calls SQLSetDescField multiple times to assign values to the following fields of the ARD:

   - Sets SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE to the value of TargetType, except that if TargetType is one of the concise identifiers of a datetime or interval subtype, it sets SQL_DESC_TYPE to SQL_DATETIME or SQL_INTERVAL, respectively; sets SQL_DESC_CONCISE_TYPE to the concise identifier; and sets SQL_DESC_DATETIME_INTERVAL_CODE to the corresponding datetime or interval subcode.

   - Sets one or more of SQL_DESC_LENGTH, SQL_DESC_PRECISION, SQL_DESC_SCALE, and SQL_DESC_DATETIME_INTERVAL_PRECISION, as appropriate for TargetType.

   - Sets the SQL_DESC_OCTET_LENGTH field to the value of BufferLength.

   - Sets the SQL_DESC_DATA_PTR field to the value of TargetValue.

   - Sets the SQL_DESC_INDICATOR_PTR field to the value of StrLen_or_Ind. (See the following paragraph.)

   - Sets the SQL_DESC_OCTET_LENGTH_PTR field to the value of StrLen_or_Ind. (See the following paragraph.)

The variable that the StrLen_or_Ind argument refers to is used for both indicator and length information. If a fetch encounters a null value for the column, it stores SQL_NULL_DATA in this variable; otherwise, it stores the data length in this variable. Passing a null pointer as StrLen_or_Ind keeps the fetch operation from returning the data length but makes the fetch fail if it encounters a null value and has no way to return SQL_NULL_DATA.

If the call to SQLBindCol fails, the content of the descriptor fields that it would have set in the ARD are undefined and the value of the SQL_DESC_COUNT field of the ARD is unchanged.

Implicit Resetting of COUNT Field

SQLBindCol sets SQL_DESC_COUNT to the value of the ColumnNumber argument only when this would increase the value of SQL_DESC_COUNT. If the value in the TargetValuePtr argument is a null pointer and the value in the ColumnNumber argument is equal to SQL_DESC_COUNT (that is, when unbinding the highest bound column), then SQL_DESC_COUNT is set to the number of the highest
Cautions Regarding SQL_DEFAULT

To retrieve column data successfully, the application must determine correctly the length and starting point of the data in the application buffer. When the application specifies an explicit TargetType, application misconceptions are easily detected. However, when the application specifies a TargetType of SQL_DEFAULT, SQLBindCol can be applied to a column of a different data type from the one intended by the application, either from changes to the metadata or by applying the code to a different column. In this case, the application may not always determine the start or length of the fetched column data. This may lead to unreported data errors or memory violations.

Code Example

In the following example, an application executes a SELECT statement on the Customers table to return a result set of the customer IDs, names, and phone numbers, sorted by name. It then calls SQLBindCol to bind the columns of data to local buffers. Finally, the application fetches each row of data with SQLFetch and prints each customer's name, ID, and phone number.

For more code examples, see SQLBulkOperations Function, SQLColumns Function, SQLFetchScroll Function, and SQLSetPos Function.

```c
// SQLBindCol_ref.cpp
#include <windows.h>
#include <stdio.h>
#define UNICODE
#include <sqlext.h>
#define NAME_LEN 50
#define PHONE_LEN 20

void show_error() {
    printf("error\n");
}

int main() {
    SQLHENV henv;
    SQLDBC hdbc;
    SQLSTMT hstmt = 0;
    SQLRETURN retcode;
    SQLWCHAR szName[NAME_LEN], szPhone[PHONE_LEN], sCustID[NAME_LEN];
    SQLLEN cbName = 0, cbCustID = 0, cbPhone = 0;

    // Allocate environment handle
    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
        retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)SQL_OV_ODBC3, 0);
```

remaining bound column.
// Allocate connection handle
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
}

// Set login timeout to 5 seconds
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
}

// Connect to data source
retcode = SQLConnect(hdbc, (SQLWCHAR*) L"NorthWind", SQL_NTS, (SQLWCHAR*) NULL);

// Allocate statement handle
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

    retcode = SQLExecDirect(hstmt, (SQLWCHAR *) L"SELECT CustomerID, ContactName, Phone FROM CUSTOMERS ORDER BY 2, 1, 3", SQL_NTS);
}

// Bind columns 1, 2, and 3
retcode = SQLBindCol(hstmt, 1, SQL_C_CHAR, &sCustID, 100, &cbCustID);
retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, szName, NAME_LEN, &cbName);
retcode = SQLBindCol(hstmt, 3, SQL_C_CHAR, szPhone, PHONE_LEN, &cbPhone);

// Fetch and print each row of data. On an error, display a message and break
for (i = 0; i < n; i++) {
    retcode = SQLFetch(hstmt);
    if (retcode == SQL_ERROR || retcode == SQL_SUCCESS_WITH_INFO)
        show_error();
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
        wprintf(L"%d: %S %S %S\n", i + 1, sCustID, szName, szPhone);
    else
        break;
}

// Process data
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    SQLCancel(hstmt);
    SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
}

SQLDisconnect(hdbc);

SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
}

Related Functions

Also see, Sample ODBC Program.
For information about | See
---|---
Returning information about a column in a result set | SQLDescribeCol Function
Fetching a block of data or scrolling through a result set | SQLFetchScroll Function
Fetching multiple rows of data | SQLFetch Function
Releasing column buffers on the statement | SQLFreeStmt Function
Fetching part or all of a column of data | SQLGetData Function
Returning the number of result set columns | SQLNumResultCols Function

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

# SQLBindParameter Function

## Conformance

Version Introduced: ODBC 2.0 Standards Compliance: ODBC

## Summary

`SQLBindParameter` binds a buffer to a parameter marker in an SQL statement. `SQLBindParameter` supports binding to a Unicode C data type, even if the underlying driver does not support Unicode data.

**Note**

This function replaces the ODBC 1.0 function **SQLSetParam**. For more information, see "Comments."

## Syntax

```c
SQLRETURN SQLBindParameter(
    SQLHSTMT StatementHandle,
    SQLUSMALLINT ParameterNumber,
    SQLSMALLINT InputOutputType,
    SQLSMALLINT ValueType,
    SQLPOINTER Value,
    SQLSMALLINT BufferLength,
    SQLSMALLINT Size
);
```
Arguments

**StatementHandle**
[Input] Statement handle.

**ParameterNumber**
[Input] Parameter number, ordered sequentially in increasing parameter order, starting at 1.

**InputOutputType**
[Input] The type of the parameter. For more information, see "InputOutputType Argument" in "Comments."

**ValueType**
[Input] The C data type of the parameter. For more information, see "ValueType Argument" in "Comments."

**ParameterType**
[Input] The SQL data type of the parameter. For more information, see "ParameterType Argument" in "Comments."

**ColumnSize**
[Input] The size of the column or expression of the corresponding parameter marker. For more information, see "ColumnSize Argument" in "Comments."

If your application will run on a 64-bit Windows operating system, see [ODBC 64-Bit Information](#).

**DecimalDigits**
[Input] The decimal digits of the column or expression of the corresponding parameter marker. For more information about column size, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size.

**ParameterValuePtr**
[Deferred Input] A pointer to a buffer for the parameter's data. For more information, see "ParameterValuePtr Argument" in "Comments."

**BufferLength**
[Input/Output] Length of the ParameterValuePtr buffer in bytes. For more information, see "BufferLength Argument" in "Comments."

See [ODBC 64-Bit Information](#), if your application will run on a 64-bit operating system.

**StrLen_or_IndPtr**
[Deferred Input] A pointer to a buffer for the parameter's length. For more information, see "StrLen_or_IndPtr Argument" in "Comments."

Returns

```c
SQLSMALLINT ParameterType,
SQLULEN ColumnSize,
SQLSMALLINT DecimalDigits,
SQLPOINTER ParameterValuePtr,
SQLLEN BufferLength,
SQLLEN * StrLen_or_IndPtr);
```
When `SQLBindParameter` returns `SQL_ERROR` or `SQL_SUCCESS_WITH_INFO`, an associated SQLSTATE value can be obtained by calling `SQLGetDiagRec` with a `HandleType` of `SQL_HANDLE_STMT` and a `Handle` of `StatementHandle`. The following table lists the SQLSTATE values typically returned by `SQLBindParameter` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is `SQL_ERROR`, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns <code>SQL_SUCCESS_WITH_INFO</code>.)</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>The data type identified by the <code>ValueType</code> argument cannot be converted to the data type identified by the <code>ParameterType</code> argument. Notice that this error may be returned by <code>SQLExecDirect</code>, <code>SQLExecute</code>, or <code>SQLPutData</code> at execution time, instead of by <code>SQLBindParameter</code>.</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>(DM) The value specified for the argument <code>ParameterNumber</code> was less than 1.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <code>*MessageText</code> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY003</td>
<td>Invalid application buffer type</td>
<td>The value specified by the argument <code>ValueType</code> was not a valid C data type or <code>SQL_C_DEFAULT</code>.</td>
</tr>
<tr>
<td>HY004</td>
<td>Invalid SQL data type</td>
<td>The value specified for the argument <code>ParameterType</code> was neither a valid ODBC SQL data type identifier nor a driver-specific SQL data type identifier supported by the driver.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid argument value</td>
<td>(DM) The argument <code>ParameterValuePtr</code> was a null pointer, the argument <code>StrLen_or_IndPtr</code> was a null pointer, and the argument <code>InputOutputType</code> was not <code>SQL_PARAM_OUTPUT</code>. (DM) SQL_PARAM_OUTPUT, where the argument <code>ParameterValuePtr</code> was a null pointer, the C type was char or binary, and the BufferLength (<code>cbValueMax</code>) was greater than 0.</td>
</tr>
</tbody>
</table>
| HY010 | Function sequence error | (DM) An asynchronously executing function was called for the connection handle that is associated with the `StatementHandle`. This asynchronous function was still executing when `SQLBindParameter` was called.

(DM) **SQLExecute**, **SQLExecDirect**, or **SQLMoreResults** was called for the `StatementHandle` and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.

(DM) An asynchronously executing function was called for the `StatementHandle` and was still executing when this function was called.

(DM) **SQLExecute**, **SQLExecDirect**, **SQLBulkOperations**, or **SQLSetPos** was called for the `StatementHandle` and returned SQL_NEEDED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. |
| HY013 | Memory management error | The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions. |
| HY021 | Inconsistent descriptor information | The descriptor information checked during a consistency check was not consistent. (See the "Consistency Checks" section in `SQLSetDescField`.) The value specified for the argument `DecimalDigits` was outside the range of values supported by the data source for a column of the SQL data type specified by the `ParameterType` argument. |
| HY090 | Invalid string or buffer length | (DM) The value in `BufferLength` was less than 0. (See the description of the SQL_DESC_DATA_PTR field in `SQLSetDescField`.) |
| HY104 | Invalid precision or scale value | The value specified for the argument `ColumnSize` or `DecimalDigits` was outside the range of values supported by the data source for a column of the SQL data type specified by the `ParameterType` argument. |
| HY105 | Invalid parameter type | (DM) The value specified for the argument `InputOutputType` was invalid. (See "Comments." ) |
| HY117 | Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed. | (DM) For more information about suspended state, see `SQLEndTran Function`. |
| HYC00 | Optional feature not implemented | The driver or data source does not support the conversion specified by the combination of the value specified for the argument `ValueType` and the driver-specific value specified |
for the argument ParameterType.

The value specified for the argument ParameterType was a valid ODBC SQL data type identifier for the version of ODBC supported by the driver but was not supported by the driver or data source.

The driver supports only ODBC 2.x and the argument ValueType was one of the following:

SQL_C_NUMERIC SQL_C_SBIGINT SQL_C_UBIGINT

and all the interval C data types listed in C Data Types in Appendix D: Data Types.

The driver only supports ODBC versions prior to 3.50, and the argument ValueType was SQL_C_GUID.

| HYT01 | Connection timeout expired | The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT. |
| IM001 | Driver does not support this function | (DM) The driver associated with the StatementHandle does not support the function. |

Comments

An application calls SQLBindParameter to bind each parameter marker in an SQL statement. Bindings remain in effect until the application calls SQLBindParameter again, calls SQLFreeStmt with the SQL_RESET_PARAMS option, or calls SQLSetDescField to set the SQL_DESC_COUNT header field of the APD to 0.

For more information about parameters, see Statement Parameters. For more information about parameter data types and parameter markers, see Parameter Data Types and Parameter Markers in Appendix C: SQL Grammar.

ParameterNumber Argument

If ParameterNumber in the call to SQLBindParameter is greater than the value of SQL_DESC_COUNT, SQLSetDescField is called to increase the value of SQL_DESC_COUNT to ParameterNumber.

InputOutputType Argument

The InputOutputType argument specifies the type of the parameter. This argument sets the SQL_DESC_PARAMETER_TYPE field of the IPD. All parameters in SQL statements that do not call procedures, such as INSERT statements, are input parameters. Parameters in procedure calls can be input, input/output, or output parameters. (An application calls SQLProcedureColumns to determine
the type of a parameter in a procedure call; parameters whose type cannot be determined are assumed to be input parameters.)

The **InputOutputType** argument is one of the following values:

- **SQL_PARAM_INPUT.** The parameter marks a parameter in an SQL statement that does not call a procedure, such as an **INSERT** statement, or it marks an input parameter in a procedure. For example, the parameters in **INSERT INTO Employee VALUES (?, ?, ?)** are input parameters, whereas the parameters in `{call AddEmp(?, ?, ?)}` can be, but are not necessarily, input parameters.

  When the statement is executed, the driver sends data for the parameter to the data source; the ***ParameterValuePtr** buffer must contain a valid input value, or the ***StrLen_or_IndPtr** buffer must contain SQL_NULL_DATA, SQL_DATA_AT_EXEC, or the result of the SQL_LEN_DATA_AT_EXEC macro.

  If an application cannot determine the type of a parameter in a procedure call, it sets **InputOutputType** to SQL_PARAM_INPUT; if the data source returns a value for the parameter, the driver discards it.

- **SQL_PARAM_INPUT_OUTPUT.** The parameter marks an input/output parameter in a procedure. For example, the parameter in `{call GetEmpDept(?)}` is an input/output parameter that accepts an employee's name and returns the name of the employee's department.

  When the statement is executed, the driver sends data for the parameter to the data source; the ***ParameterValuePtr** buffer must contain a valid input value, or the ***StrLen_or_IndPtr** buffer must contain SQL_NULL_DATA, SQL_DATA_AT_EXEC, or the result of the SQL_LEN_DATA_AT_EXEC macro. After the statement is executed, the driver returns data for the parameter to the application; if the data source does not return a value for an input/output parameter, the driver sets the ***StrLen_or_IndPtr** buffer to SQL_NULL_DATA.

  **Note**

  When an ODBC 1.0 application calls **SQLSetParam** in an ODBC 2.0 driver, the Driver Manager converts this to a call to **SQLBindParameter** in which the **InputOutputType** argument is set to SQL_PARAM_INPUT_OUTPUT.

- **SQL_PARAM_OUTPUT.** The parameter marks the return value of a procedure or an output parameter in a procedure; in either case, these are known as output parameters. For example, the parameter in `{?=call GetNextEmpID}` is an output parameter that returns the next employee ID.

  After the statement is executed, the driver returns data for the parameter to the application, unless the **ParameterValuePtr** and **StrLen_or_IndPtr** arguments are both null pointers, in which case the driver discards the output value. If the data source does not return a value for an output parameter, the driver sets the ***StrLen_or_IndPtr** buffer to SQL_NULL_DATA.

- **SQL_PARAM_INPUT_OUTPUT_STREAM.** Indicates that an input/output parameter should be streamed. **SQLGetData** can read parameter values in parts. **BufferLength** is ignored because the buffer length will be determined at the call of **SQLGetData**. The value of the **StrLen_or_IndPtr** buffer must contain SQL_NULL_DATA, SQL_DEFAULT_PARAM, SQL_DATA_AT_EXEC, or the result of the SQL_LEN_DATA_AT_EXEC macro. A parameter must be bound as a data-at-execution (DAE) parameter at input if it will be streamed at output. **ParameterValuePtr** can be any non-null pointer value that will be returned by **SQLParamData** as the user-defined token whose value was passed with **ParameterValuePtr** for both input and output. For more information, see Retrieving Output Parameters Using SQLGetData.
- SQL_PARAM_OUTPUT_STREAM. Same as SQL_PARAM_INPUT_OUTPUT_STREAM, for an output parameter. *StrLen_or_IndPtr is ignored at input.

The following table lists different combinations of InputOutputType and *StrLen_or_IndPtr:

<table>
<thead>
<tr>
<th>InputOutputType</th>
<th>*StrLen_or_IndPtr</th>
<th>Outcome</th>
<th>Remark on ParameterValuePtr</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_PARAM_INPUT</td>
<td>SQL_LEN_DATA_AT_EXEC(len) or SQL_DATA_AT_EXEC</td>
<td>Input in parts</td>
<td>ParameterValuePtr can be any pointer value that will be returned by SQLParamData as the user-defined token whose value was passed with ParameterValuePtr.</td>
</tr>
<tr>
<td>SQL_PARAM_INPUT</td>
<td>Not SQL_LEN_DATA_AT_EXEC(len) or SQL_DATA_AT_EXEC</td>
<td>Input bound buffer</td>
<td>ParameterValuePtr is the address of the input buffer.</td>
</tr>
<tr>
<td>SQL_PARAM_OUTPUT</td>
<td>Ignored at input.</td>
<td>Output bound buffer</td>
<td>ParameterValuePtr is the address of the output buffer.</td>
</tr>
<tr>
<td>SQL_PARAM_OUTPUT_STREAM</td>
<td>Ignored at input.</td>
<td>Streamed output</td>
<td>ParameterValuePtr can be any pointer value, which will be returned by SQLParamData as the user-defined token whose value was passed with ParameterValuePtr.</td>
</tr>
<tr>
<td>SQL_PARAM_INPUT_OUTPUT</td>
<td>SQL_LEN_DATA_AT_EXEC(len) or SQL_DATA_AT_EXEC</td>
<td>Input in parts and output bound buffer</td>
<td>ParameterValuePtr is the address of the output buffer, which will also be returned by SQLParamData as the user-defined token whose value was passed with ParameterValuePtr.</td>
</tr>
<tr>
<td>SQL_PARAM_INPUT_OUTPUT</td>
<td>Not SQL_LEN_DATA_AT_EXEC(len) or SQL_DATA_AT_EXEC</td>
<td>Input bound buffer and output bound buffer</td>
<td>ParameterValuePtr is the address of the shared input/output buffer.</td>
</tr>
</tbody>
</table>
The driver must decide which SQL types are allowed when an application binds an output or input-output parameter as streamed. The driver manager will not generate an error for an invalid SQL type.

**ValueType Argument**

The **ValueType** argument specifies the C data type of the parameter. This argument sets the SQL_DESC_TYPE, SQL_DESC_CONCISE_TYPE, and SQL_DESC_DATETIME_INTERVAL_CODE fields of the APD. This must be one of the values in the **C Data Types** section of Appendix D: Data Types.

If the **ValueType** argument is one of the interval data types, the SQL_DESC_TYPE field of the ParameterNumber record of the APD is set to SQL_INTERVAL, the SQL_DESC_CONCISE_TYPE field of the APD is set to the concise interval data type, and the SQL_DESC_DATETIME_INTERVAL_CODE field of the ParameterNumber record is set to a subcode for the specific interval data type. (See Appendix D: Data Types.) The default interval leading precision (2) and default interval seconds precision (6), as set in the SQL_DESC_DATETIME_INTERVAL_PRECISION and SQL_DESC_PRECISION fields of the APD, respectively, are used for the data. If either default precision is not appropriate, the application should explicitly set the descriptor field by a call to SQLSetDescField or SQLSetDescRec.

If the **ValueType** argument is one of the datetime data types, the SQL_DESC_TYPE field of the ParameterNumber record of the APD is set to SQL_DATETIME, the SQL_DESC_CONCISE_TYPE field of the ParameterNumber record of the APD is set to the concise datetime C data type, and the SQL_DESC_DATETIME_INTERVAL_CODE field of the ParameterNumber record is set to a subcode for the specific datetime data type. (See Appendix D: Data Types.)

If the **ValueType** argument is an SQL_C_NUMERIC data type, the default precision (which is driver-defined) and the default scale (0), as set in the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the APD, are used for the data. If the default precision or scale is not appropriate, the application should explicitly set the descriptor field by a call to SQLSetDescField or SQLSetDescRec.

SQL_C_DEFAULT specifies that the parameter value be transferred from the default C data type for the SQL data type specified with **ParameterType**.

You can also specify an extended C data type. For more information, see **C Data Types in ODBC**.

For more information, see Default C Data Types, Converting Data from C to SQL Data Types, and Converting Data from SQL to C Data Types in Appendix D: Data Types.
### ParameterType Argument

This must be one of the values listed in the SQL Data Types section of Appendix D: Data Types, or it must be a driver-specific value. This argument sets the SQL_DESC_TYPE, SQL_DESC_CONCISE_TYPE, and SQL_DESC_DATETIME_INTERVAL_CODE fields of the IPD.

If the ParameterType argument is one of the datetime identifiers, the SQL_DESC_TYPE field of the IPD is set to SQL_DATETIME, the SQL_DESC_CONCISE_TYPE field of the IPD is set to the concise datetime SQL data type, and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to the appropriate datetime subcode value.

If ParameterType is one of the interval identifiers, the SQL_DESC_TYPE field of the IPD is set to SQL_INTERVAL, the SQL_DESC_CONCISE_TYPE field of the IPD is set to the concise SQL interval data type, and the SQL_DESC_DATETIME_INTERVAL_CODE field of the IPD is set to the appropriate interval subcode. The SQL_DESC_DATETIME_INTERVAL_PRECISION field of the IPD is set to the interval leading precision, and the SQL_DESC_PRECISION field is set to the interval seconds precision, if applicable. If the default value of SQL_DESC_DATETIME_INTERVAL_PRECISION or SQL_DESC_PRECISION is not appropriate, the application should explicitly set it by calling SQLSetDescField. For more information about any of these fields, see SQLSetDescField.

If the ValueType argument is a SQL_NUMERIC data type, the default precision (which is driver-defined) and the default scale (0), as set in the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the IPD, are used for the data. If the default precision or scale is not appropriate, the application should explicitly set the descriptor field by a call to SQLSetDescField or SQLSetDescRec.

For information about how data is converted, see Converting Data from C to SQL Data Types and Converting Data from SQL to C Data Types in Appendix D: Data Types.

### ColumnSize Argument

The ColumnSize argument specifies the size of the column or expression that corresponds to the parameter marker, the length of that data, or both. This argument sets different fields of the IPD, depending on the SQL data type (the ParameterType argument). The following rules apply to this mapping:

- If ParameterType is SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR, SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY, or one of the concise SQL datetime or interval data types, the SQL_DESC_LENGTH field of the IPD is set to the value of ColumnSize. (For more information, see the Column Size, Decimal Digits, Transfer Octet Length, and Display Size section in Appendix D: Data Types.)

- If ParameterType is SQL_DECIMAL, SQL_NUMERIC, SQL_FLOAT, SQL_REAL, or SQL_DOUBLE, the SQL_DESC_PRECISION field of the IPD is set to the value of ColumnSize.

- For other data types, the ColumnSize argument is ignored.

For more information, see "Passing Parameter Values" and SQL_DATA_AT_EXEC in "StrLen_or_IndPtr Argument."

### DecimalDigits Argument
If ParameterType is SQL_TYPE_TIME, SQL_TYPE_TIMESTAMP, SQL_INTERVAL_SECOND, SQL_INTERVAL_DAY_TO_SECOND, SQL_INTERVAL_HOUR_TO_SECOND, or SQL_INTERVAL_MINUTE_TO_SECOND, the SQL_DESC_PRECISION field of the IPD is set to DecimalDigits. If ParameterType is SQL_NUMERIC or SQL_DECIMAL, the SQL_DESC_SCALE field of the IPD is set to DecimalDigits. For all other data types, the DecimalDigits argument is ignored.

ParameterValuePtr Argument

The ParameterValuePtr argument points to a buffer that, when SQLExecute or SQLExecDirect is called, contains the actual data for the parameter. The data must be in the form specified by the ValueType argument. This argument sets the SQL_DESC_DATA_PTR field of the APD. An application can set the ParameterValuePtr argument to a null pointer, as long as *StrLen_or_IndPtr is SQL_NULL_DATA or SQL_DATA_AT_EXEC. (This applies only to input or input/output parameters.)

If *StrLen_or_IndPtr is the result of the SQL_LEN_DATA_AT_EXEC(length) macro or SQL_DATA_AT_EXEC, then ParameterValuePtr is an application-defined pointer value that is associated with the parameter. It is returned to the application through SQLParamData. For example, ParameterValuePtr might be a non-zero token such as a parameter number, a pointer to data, or a pointer to a structure that the application used to bind input parameters. However, note that if the parameter is an input/output parameter, ParameterValuePtr must be a pointer to a buffer where the output value will be stored. If the value in the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1, the application can use the value pointed to by the SQL_ATTR_PARAMS_PROCESSED_PTR statement attribute together with the ParameterValuePtr argument. For example, ParameterValuePtr might point to an array of values and the application might use the value pointed to by SQL_ATTR_PARAMS_PROCESSED_PTR to retrieve the correct value from the array. For more information, see "Passing Parameter Values" later in this section.

If the InputOutputType argument is SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT, ParameterValuePtr points to a buffer in which the driver returns the output value. If the procedure returns one or more result sets, the *ParameterValuePtr buffer is not guaranteed to be set until all result sets/row counts have been processed. If the buffer is not set until processing is complete, the output parameters and return values are unavailable until SQLMoreResults returns SQL_NO_DATA. Calling SQLCloseCursor or SQLFreeStmt with an Option of SQL_CLOSE will cause these values to be discarded.

If the value in the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1, ParameterValuePtr points to an array. A single SQL statement processes the complete array of input values for an input or input/output parameter and returns an array of output values for an input/output or output parameter.

BufferLength Argument

For character and binary C data, the BufferLength argument specifies the length of the *ParameterValuePtr buffer (if it is a single element) or the length of an element in the *ParameterValuePtr array (if the value in the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1). This argument sets the SQL_DESC_OCTET_LENGTH record field of the APD. If the application specifies multiple values, BufferLength is used to determine the location of values in the *ParameterValuePtr array, both on input and on output. For input/output and output parameters, it is used to determine whether to truncate character and binary C data on output:

- For character C data, if the number of bytes available to return is greater than or equal to
BufferLength, the data in *ParameterValuePtr is truncated to BufferLength less the length of a null-termination character and is null-terminated by the driver.

- For binary C data, if the number of bytes available to return is greater than BufferLength, the data in *ParameterValuePtr is truncated to BufferLength bytes.

For all other types of C data, the BufferLength argument is ignored. The length of the *ParameterValuePtr buffer (if it is a single element) or the length of an element in the *ParameterValuePtr array (if the application calls SQLSetStmtAttr with an Attribute argument of SQL_ATTR_PARAMSET_SIZE to specify multiple values for each parameter) is assumed to be the length of the C data type.

For streamed output or streamed input/output parameters, the BufferLength argument is ignored because the buffer length is specified in SQLGetData.

---

**Note**

When an ODBC 1.0 application calls SQLSetParam in an ODBC 3.x driver, the Driver Manager converts this to a call to SQLBindParameter in which the BufferLength argument is always SQL_SETPARAM_VALUE_MAX. Because the Driver Manager returns an error if an ODBC 3.x application sets BufferLength to SQL_SETPARAM_VALUE_MAX, an ODBC 3.x driver can use this to determine when it is called by an ODBC 1.0 application.

---

**Note**

In SQLSetParam, the way in which an application specifies the length of the *ParameterValuePtr buffer so that the driver can return character or binary data, and the way in which an application sends an array of character or binary parameter values to the driver, are driver-defined.

---

**StrLen_or_IndPtr Argument**

The StrLen_or_IndPtr argument points to a buffer that, when SQLExecute or SQLExecDirect is called, contains one of the following. (This argument sets the SQL_DESC_OCTET_LENGTH_PTR and SQL_DESC_INDICATOR_PTR record fields of the application parameter pointers.)

- The length of the parameter value stored in *ParameterValuePtr. This is ignored except for character or binary C data.
- SQL_NTS. The parameter value is a null-terminated string.
- SQL_NULL_DATA. The parameter value is NULL.
- SQL_DEFAULT_PARAM. A procedure is to use the default value of a parameter, instead of a value retrieved from the application. This value is valid only in a procedure called in ODBC canonical syntax, and then only if the InputOutputType argument is SQL_PARAM_INPUT, SQL_PARAM_INPUT_OUTPUT, or SQL_PARAM_INPUT_OUTPUT_STREAM. When *StrLen_or_IndPtr is SQL_DEFAULT_PARAM, the ValueType, ParameterType, ColumnSize, DecimalDigits, BufferLength, and ParameterValuePtr arguments are ignored for input parameters and are used only to define the output parameter value for input/output parameters.
- The result of the SQL_LEN_DATA_AT_EXEC(length) macro. The data for the parameter will be sent with SQLPutData. If the ParameterType argument is SQL_LONGVARBINARY,
SQL_LONGVARCHAR, or a long, data source–specific data type, and the driver returns "Y" for the SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo, length is the number of bytes of data to be sent for the parameter; otherwise, length must be a nonnegative value and is ignored. For more information, see "Passing Parameter Values," later in this section.

For example, to specify that 10,000 bytes of data will be sent with SQLPutData in one or more calls, for an SQL_LONGVARCHAR parameter, an application sets *StrLen_or_IndPtr to SQL_LEN_DATA_AT_EXEC(10000).

- SQL_DATA_AT_EXEC. The data for the parameter will be sent with SQLPutData. This value is used by ODBC 1.0 applications when they call ODBC 3.x drivers. For more information, see "Passing Parameter Values," later in this section.

If StrLen_or_IndPtr is a null pointer, the driver assumes that all input parameter values are non-NULL and that character and binary data is null-terminated. If InputOutputType is SQL_PARAM_OUTPUT or SQL_PARAM_OUTPUT_STREAM and ParameterValuePtr and StrLen_or_IndPtr are both null pointers, the driver discards the output value.

**Note**

Application developers are strongly discouraged from specifying a null pointer for StrLen_or_IndPtr when the data type of the parameter is SQL_C_BINARY. To make sure that a driver does not unexpectedly truncate SQL_C_BINARY data, StrLen_or_IndPtr should contain a pointer to a valid length value.

If the InputOutputType argument is SQL_PARAM_INPUT_OUTPUT, SQL_PARAM_OUTPUT, SQL_PARAM_INPUT_OUTPUT_STREAM, or SQL_PARAM_OUTPUT_STREAM, StrLen_or_IndPtr points to a buffer in which the driver returns SQL_NULL_DATA, the number of bytes available to return in *ParameterValuePtr (excluding the null-termination byte of character data), or SQL_NO_TOTAL (if the number of bytes available to return cannot be determined). If the procedure returns one or more result sets, the *StrLen_or_IndPtr buffer is not guaranteed to be set until all results have been fetched.

If the value in the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1, StrLen_or_IndPtr points to an array of SQLLEN values. These can be any of the values listed earlier in this section and are processed with a single SQL statement.

### Passing Parameter Values

An application can pass the value for a parameter either in the *ParameterValuePtr buffer or with one or more calls to SQLPutData. Parameters whose data is passed with SQLPutData are known as data-at-execution parameters. These are typically used to send data for SQL_LONGVARBINARY and SQL_LONGVARCHAR parameters, and can be mixed with other parameters.

To pass parameter values, an application performs the following sequence of steps:

1. Calls SQLBindParameter for each parameter to bind buffers for the parameter's value (*ParameterValuePtr argument) and length/indicator (*StrLen_or_IndPtr argument). For data-at-execution parameters, ParameterValuePtr is an application-defined pointer value such as a parameter number or a pointer to data. The value will be returned later and can be used to identify the parameter.

2. Places values for input and input/output parameters in the *ParameterValuePtr and
*StrLen_or_IndPtr buffers:

- For normal parameters, the application places the parameter value in the *ParameterValuePtr buffer and the length of that value in the *StrLen_or_IndPtr buffer. For more information, see Setting Parameter Values.

- For data-at-execution parameters, the application puts the result of the SQL_LEN_DATA_AT_EXEC(length) macro (when calling an ODBC 2.0 driver) in the *StrLen_or_IndPtr buffer.

3. Calls SQLExecute or SQLExecDirect to execute the SQL statement.

- If there are no data-at-execution parameters, the process is complete.

- If there are any data-at-execution parameters, the function returns SQL_NEED_DATA.

4. Calls SQLParamData to retrieve the application-defined value specified in the ParameterValuePtr argument of SQLBindParameter for the first data-at-execution parameter to be processed. SQLParamData returns SQL_NEED_DATA.

Note

Although data-at-execution parameters resemble data-at-execution columns, the value returned by SQLParamData is different for each. Data-at-execution parameters are parameters in an SQL statement for which data will be sent with SQLPutData when the statement is executed with SQLExecDirect or SQLExecute. They are bound with SQLBindParameter. The value returned by SQLParamData is a pointer value passed to SQLBindParameter in the ParameterValuePtr argument. Data-at-execution columns are columns in a rowset for which data will be sent with SQLPutData when a row is updated or added with SQLBulkOperations or updated with SQLSetPos. They are bound with SQLBindCol. The value returned by SQLParamData is the address of the row in the *TargetValuePtr buffer (set by a call to SQLBindCol) that is being processed.

5. Calls SQLPutData one or more times to send data for the parameter. More than one call is needed if the data value is larger than the *ParameterValuePtr buffer specified in SQLPutData; multiple calls to SQLPutData for the same parameter are allowed only when sending character C data to a column with a character, binary, or data source–specific data type or when sending binary C data to a column with a character, binary, or data source–specific data type.

6. Calls SQLParamData again to signal that all data has been sent for the parameter.

- If there are more data-at-execution parameters, SQLParamData returns SQL_NEED_DATA and the application-defined value for the next data-at-execution parameter to be processed. The application repeats steps 4 and 5.

- If there are no more data-at-execution parameters, the process is complete. If the statement was successfully executed, SQLParamData returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO; if the execution failed, it returns SQL_ERROR. At this point, SQLParamData can return any SQLSTATE that can be returned by the function that is used to execute the statement (SQLExecDirect or SQLExecute).

Output values for any input/output or output parameters are available in the *ParameterValuePtr and *StrLen_or_IndPtr buffers after the application retrieves all result sets generated by the statement.

Calling SQLExecute or SQLExecDirect puts the statement in an SQL_NEED_DATA state. At this point, the application can call only SQLCancel, SQLGetDiagField, SQLGetDiagRec,
**Retrieving Streamed Output Parameters**

When an application sets `InputOutputType` to `SQL_PARAM_INPUT_OUTPUT_STREAM` or `SQL_PARAM_OUTPUT_STREAM`, the output parameter value must be retrieved by one or more calls to `SQLGetData`. When the driver has a streamed output parameter value to return to the application, it will return `SQL_PARAM_DATA_AVAILABLE` in response to a call to the following functions: `SQLMoreResults`, `SQLExecute`, and `SQLExecDirect`. An application calls `SQLParamData` to determine which parameter value is available.

For more information about `SQL_PARAM_DATA_AVAILABLE` and streamed output parameters, see Retrieving Output Parameters Using `SQLGetData`.

**Using Arrays of Parameters**

When an application prepares a statement with parameter markers and passes in an array of parameters, there are two different ways this can be executed. One way is for the driver to rely on the array-processing capabilities of the back end, in which case the whole statement with the array of parameters is treated as one atomic unit. Oracle is an example of a data source that supports array processing capabilities. Another way to implement this feature is for the driver to generate a batch of SQL statements, one SQL statement for each set of parameters in the parameter array, and execute the batch. Arrays of parameters cannot be used with an `UPDATE WHERE CURRENT OF` statement.

When an array of parameters is processed, individual result sets/row counts (one for each parameter set) can be available or result sets/rows counts can be rolled up into one. The `SQL_PARAM_ARRAY_ROW_COUNTS` option in `SQLGetInfo` indicates whether row counts are available for each set of parameters (SQL_PARC_BATCH) or only one row count is available (SQL_PARC_NO_BATCH).

The `SQL_PARAM_ARRAY_SELECTS` option in `SQLGetInfo` indicates whether a result set is available for each set of parameters (SQL_PAS_BATCH) or only one result set is available (SQL_PAS_NO_SELECT). If the driver does not allow a result set–generating statement to be executed with an array of parameters, `SQL_PARAM_ARRAY_SELECTS` returns SQL_PAS_NO_SELECT.

For more information, see `SQLGetInfo Function`.

To support arrays of parameters, the `SQL_ATTR_PARAMSET_SIZE` statement attribute is set to specify the number of values for each parameter. If the field is greater than 1, the `SQL_DESC_DATA_PTR`, `SQL_DESC_INDICATOR_PTR`, and `SQL_DESC_OCTET_LENGTH_PTR` fields of the APD must point to arrays. The cardinality of each array is equal to the value of `SQL_ATTR_PARAMSET_SIZE`.

The `SQL_DESC_ROWS_PROCESSED_PTR` field of the APD points to a buffer that contains the number of sets of parameters that have been processed, including error sets. As each set of parameters is processed, the driver stores a new value in the buffer. No number will be returned if this is a null
pointer. When arrays of parameters are used, the value pointed to by the
SQL_DESC_ROWS_PROCESSED_PTR field of the APD is populated even if SQL_ERROR is returned by
the setting function. If SQL_NEED_DATA is returned, the value pointed to by the
SQL_DESC_ROWS_PROCESSED_PTR field of the APD is set to the set of parameters that is being
processed.

What occurs when an array of parameters is bound and an UPDATE WHERE CURRENT OF statement
is executed is driver-defined.

Column–Wise Parameter Binding

In column-wise binding, the application binds separate parameter and length/indicator arrays to each
parameter.

To use column-wise binding, the application first sets the SQL_ATTR_PARAM_BIND_TYPE statement
attribute to SQL_PARAM_BIND_BY_COLUMN. (This is the default.) For each column to be bound, the
application performs the following steps:

1. Allocates a parameter buffer array.
2. Allocates an array of length/indicator buffers.

Note

If the application writes directly to descriptors when column-wise binding is used, separate
arrays can be used for length and indicator data.

3. Calls SQLBindParameter with the following arguments:

   - ValueType is the C type of a single element in the parameter buffer array.
   - ParameterType is the SQL type of the parameter.
   - ParameterValuePtr is the address of the parameter buffer array.
   - BufferLength is the size of a single element in the parameter buffer array. The
     BufferLength argument is ignored when the data is fixed-length data.
   - StrLen_or_IndPtr is the address of the length/indicator array.

For more information about how this information is used, see "ParameterValuePtr Argument" in
"Comments," later in this section. For more information about column-wise binding of parameters, see
Binding Arrays of Parameters.

Row–Wise Parameter Binding

In row-wise binding, the application defines a structure that contains parameter and length/indicator
buffers for each parameter to be bound.

To use row-wise binding, the application performs the following steps:
1. Defines a structure to hold a single set of parameters (including both parameter and length/indicator buffers) and allocates an array of these structures.

**Note**

If the application writes directly to descriptors when row-wise binding is used, separate fields can be used for length and indicator data.

2. Sets the SQL_ATTR_PARAM_BIND_TYPE statement attribute to the size of the structure that contains a single set of parameters or to the size of an instance of a buffer into which the parameters will be bound. The length must include space for all the bound parameters, and any padding of the structure or buffer, to make sure that when the address of a bound parameter is incremented with the specified length, the result will point to the beginning of the same parameter in the next row. When you use the `sizeof` operator in ANSI C, this behavior is guaranteed.

3. Calls `SQLBindParameter` with the following arguments for each parameter to be bound:

   - *ValueType* is the type of the parameter buffer member to be bound to the column.
   - *ParameterType* is the SQL type of the parameter.
   - *ParameterValuePtr* is the address of the parameter buffer member in the first array element.
   - *BufferLength* is the size of the parameter buffer member.
   - *StrLen_or_IndPtr* is the address of the length/indicator member to be bound.

For more information about how this information is used, see "ParameterValuePtr Argument," later in this section. For more information about row-wise binding of parameters, see the Binding Arrays of Parameters.

**Error Information**

If a driver does not implement parameter arrays as batches (the SQL_PARAM_ARRAY_ROW_COUNTS option is equal to SQL_PARC_NO_BATCH), error situations are handled as if one statement were executed. If the driver does implement parameter arrays as batches, an application can use the SQL_DESC_ARRAY_STATUS_PTR header field of the IPD to determine which parameter of an SQL statement or which parameter in an array of parameters caused `SQLExecDirect` or `SQLExecute` to return an error. This field contains status information for each row of parameter values. If the field indicates that an error has occurred, fields in the diagnostic data structure will indicate the row and parameter number of the parameter that failed. The number of elements in the array will be defined by the SQL_DESC_ARRAY_SIZE header field in the APD, which can be set by the SQL_ATTR_PARAMSET_SIZE statement attribute.

**Note**

The SQL_DESC_ARRAY_STATUS_PTR header field in the APD is used to ignore parameters. For more information about ignoring parameters, see the next section, "Ignoring a Set of Parameters."

When `SQLExecute` or `SQLExecDirect` returns SQL_ERROR, the elements in the array pointed to by the SQL_DESC_ARRAY_STATUS_PTR field in the IPD will contain SQL_PARAM_ERROR, SQL_PARAM_SUCCESS, SQL_PARAM_SUCCESS_WITH_INFO, SQL_PARAM_UNUSED, or
For each element in this array, the diagnostic data structure contains one or more status records. The SQL_DIAG_ROW_NUMBER field of the structure indicates the row number of the parameter values that caused the error. If it is possible to determine the particular parameter in a row of parameters that caused the error, the parameter number will be entered in the SQL_DIAG_COLUMN_NUMBER field.

SQL_PARAM_UNUSED is entered when a parameter has not been used because an error occurred in an earlier parameter that forced SQLExecute or SQLExecDirect to abort. For example, if there are 50 parameters and an error occurred while executing the fortieth set of parameters that caused SQLExecute or SQLExecDirect to abort, then SQL_PARAM_UNUSED is entered in the status array for parameters 41 through 50.

SQL_PARAM_DIAG_UNAVAILABLE is entered when the driver treats arrays of parameters as a monolithic unit, so it does not generate this individual parameter level of error information.

Some errors in the processing of a single set of parameters cause processing of the subsequent sets of parameters in the array to stop. Other errors do not affect the processing of subsequent parameters. Which errors will stop processing is driver-defined. If processing is not stopped, all parameters in the array are processed, SQL_SUCCESS_WITH_INFO is returned as a result of the error, and the buffer defined by SQL_ATTR_PARAMS_PROCESSED_PTR is set to the total number of sets of parameters processed (as defined by the SQL_ATTR_PARAMSET_SIZE statement attribute), which includes error sets.

Caution

ODBC behavior when an error occurs in the processing of an array of parameters is different in ODBC 3.x than it was in ODBC 2.x. In ODBC 2.x, the function returned SQL_ERROR and processing ceased. The buffer pointed to by the pirow argument of SQLParamOptions contained the number of the error row. In ODBC 3.x, the function returns SQL_SUCCESS_WITH_INFO and processing may either stop or continue. If it continues, the buffer specified by SQL_ATTR_PARAMS_PROCESSED_PTR will be set to the value of all parameters processed, including those that resulted in an error. This change in behavior may cause problems for existing applications.

When SQLExecute or SQLExecDirect returns before completing the processing of all parameter sets in a parameter array, such as when SQL_ERROR or SQL_NEED_DATA is returned, the status array contains statuses for those parameters that have already been processed. The location pointed to by the SQL_DESC_ROWS_PROCESSED_PTR field in the IPD contains the row number in the parameter array that caused the SQL_ERROR or SQL_NEED_DATA error code. When an array of parameters is sent to a SELECT statement, the availability of status array values is driver-defined; they may be available after the statement has been executed or as result sets are fetched.

Ignoring a Set of Parameters

The SQL_DESC_ARRAY_STATUS_PTR field of the APD (as set by the SQL_ATTR_PARAM_STATUS_PTR statement attribute) can be used to indicate that a set of bound parameters in an SQL statement should be ignored. To direct the driver to ignore one or more sets of parameters during execution, an application should follow these steps:

1. Call SQLSetDescField to set the SQL_DESC_ARRAY_STATUS_PTR header field of the APD to point to an array of SQLUSMALLINT values to contain status information. This field can also be
set by calling `SQLSetStmtAttr` with an `Attribute` of `SQL_ATTR_PARAM_OPERATION_PTR`, which allows an application to set the field without obtaining a descriptor handle.

2. Set each element of the array defined by the `SQL_DESC_ARRAY_STATUS_PTR` field of the APD to one of two values:
   - `SQL_PARAM_IGNORE`, to indicate that the row is excluded from statement execution.
   - `SQL_PARAM_PROCEED`, to indicate that the row is included in statement execution.

3. Call `SQLExecDirect` or `SQLExecute` to execute the prepared statement.

The following rules apply to the array defined by the `SQL_DESC_ARRAY_STATUS_PTR` field of the APD:

- The pointer is set to null by default.
- If the pointer is null, all sets of parameters are used, as if all elements were set to `SQL_ROW_PROCEED`.
- Setting an element to `SQL_PARAM_PROCEED` does not guarantee that the operation will use that particular set of parameters.
- `SQL_PARAM_PROCEED` is defined as 0 in the header file.

An application can set the `SQL_DESC_ARRAY_STATUS_PTR` field in the APD to point to the same array as that pointed to by the `SQL_DESC_ARRAY_STATUS_PTR` field in the IRD. This is useful when binding parameters to row data. Parameters can then be ignored according to the status of the row data. In addition to `SQL_PARAM_IGNORE`, the following codes cause a parameter in an SQL statement to be ignored: `SQL_ROW_DELETED`, `SQL_ROW_UPDATED`, and `SQL_ROW_ERROR`. In addition to `SQL_PARAM_PROCEED`, the following codes cause an SQL statement to proceed: `SQL_ROW_SUCCESS`, `SQL_ROW_SUCCESS_WITH_INFO`, and `SQL_ROW_ADDED`.

Rebinding Parameters

An application can perform either of two operations to change a binding:

- Call `SQLBindParameter` to specify a new binding for a column that is already bound. The driver overwrites the old binding with the new one.
- Specify an offset to be added to the buffer address that was specified by the binding call to `SQLBindParameter`. For more information, see the next section, "Rebinding with Offsets."

Rebinding with Offsets

Rebinding of parameters is especially useful when an application has a buffer area setup that can contain many parameters but a call to `SQLExecDirect` or `SQLExecute` uses only a few of the parameters. The remaining space in the buffer area can be used for the next set of parameters by modifying the existing binding by an offset.

The `SQL_DESC_BIND_OFFSET_PTR` header field in the APD points to the binding offset. If the field is
non-null, the driver dereferences the pointer and, if none of the values in the SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR fields is a null pointer, adds the dereferenced value to those fields in the descriptor records at execution time. The new pointer values are used when the SQL statements are executed. The offset remains valid after rebinding. Because SQL_DESC_BIND_OFFSET_PTR is a pointer to the offset rather than the offset itself, an application can change the offset directly, without having to call SQLSetDescField or SQLSetDescRec to change the descriptor field. The pointer is set to null by default. The SQL_DESC_BIND_OFFSET_PTR field of the ARD can be set by a call to SQLSetDescField or by a call to SQLSetStmtAttr with a fAttribute of SQL_ATTR_PARAM_BIND_OFFSET_PTR.

The binding offset is always added directly to the values in the SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR fields. If the offset is changed to a different value, the new value is still added directly to the value in each descriptor field. The new offset is not added to the sum of the field value and any earlier offsets.

Descriptors

How a parameter is bound is determined by fields of the APDs and IPDs. The arguments in SQLBindParameter are used to set those descriptor fields. The fields can also be set by the SQLSetDescField functions, although SQLBindParameter is more efficient to use because the application does not have to obtain a descriptor handle to call SQLBindParameter.

⚠️ Caution

Calling SQLBindParameter for one statement can affect other statements. This occurs when the ARD associated with the statement is explicitly allocated and is also associated with other statements. Because SQLBindParameter modifies the fields of the APD, the modifications apply to all statements with which this descriptor is associated. If this is not the required behavior, the application should dissociate this descriptor from the other statements before it calls SQLBindParameter.

Conceptually, SQLBindParameter performs the following steps in sequence:

1. Calls SQLGetStmtAttr to obtain the APD handle.
2. Calls SQLGetDescField to get the APD’s SQL_DESC_COUNT field, and if the value of the ColumnNumber argument exceeds the value of SQL_DESC_COUNT, calls SQLSetDescField to increase the value of SQL_DESC_COUNT to ColumnNumber.
3. Calls SQLSetDescField multiple times to assign values to the following fields of the APD:
   - Sets SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE to the value of ValueType, except that if ValueType is one of the concise identifiers of a datetime or interval subtype, it sets SQL_DESC_TYPE to SQL_DATETIME or SQL_INTERVAL, respectively, sets SQL_DESC_CONCISE_TYPE to the concise identifier, and sets SQL_DESC_DATETIME_INTERVAL_CODE to the corresponding datetime or interval subcode.
   - Sets the SQL_DESC_OCTET_LENGTH field to the value of BufferLength.
   - Sets the SQL_DESC_DATA_PTR field to the value of ParameterValue.
   - Sets the SQL_DESC_OCTET_LENGTH_PTR field to the value of StrLen_or_Ind.
Sets the SQL_DESC_INDICATOR_PTR field also to the value of StrLen_or_Ind.

The StrLen_or_Ind parameter specifies both the indicator information and the length for the parameter value.

4. Calls SQLGetStmtAttr to obtain the IPD handle.

5. Calls SQLGetDescField to get the IPD's SQL_DESC_COUNT field, and if the value of the ColumnNumber argument exceeds the value of SQL_DESC_COUNT, calls SQLSetDescField to increase the value of SQL_DESC_COUNT to ColumnNumber.

6. Calls SQLSetDescField multiple times to assign values to the following fields of the IPD:

   - Sets SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE to the value of ParameterType, except that if ParameterType is one of the concise identifiers of a datetime or interval subtype, it sets SQL_DESC_TYPE to SQL_DATETIME or SQL_INTERVAL, respectively, sets SQL_DESC_CONCISE_TYPE to the concise identifier, and sets SQL_DESC_DATETIME_INTERVAL_CODE to the corresponding datetime or interval subcode.
   - Sets one or more of SQL_DESC_LENGTH, SQL_DESC_PRECISION, and SQL_DESC_DATETIME_INTERVAL_PRECISION, as appropriate for ParameterType.
   - Sets SQL_DESC_SCALE to the value of DecimalDigits.

If the call to SQLBindParameter fails, the content of the descriptor fields that it would have set in the APD are undefined, and the SQL_DESC_COUNT field of the APD is unchanged. In addition, the SQL_DESC_LENGTH, SQL_DESC_PRECISION, SQL_DESC_SCALE, and SQL_DESC_TYPE fields of the appropriate record in the IPD are undefined and the SQL_DESC_COUNT field of the IPD is unchanged.

### Conversion of Calls to and from SQLSetParam

When an ODBC 1.0 application calls SQLSetParam in an ODBC 3.x driver, the ODBC 3.x Driver Manager maps the call as shown in the following table.

<table>
<thead>
<tr>
<th>Call by ODBC 1.0 application</th>
<th>Call to ODBC 3.x driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLSetParam( StatementHandle, ParameterNumber, ValueType, ParameterType, LengthPrecision, ParameterScale, ParameterValuePtr, StrLen_or_IndPtr);</td>
<td>SQLBindParameter( StatementHandle, ParameterNumber, SQL_PARAM_INPUT_OUTPUT, ValueType, ParameterType, ColumnSize, DecimalDigits, ParameterValuePtr, SQL_SETPARAM_VALUE_MAX, StrLen_or_IndPtr);</td>
</tr>
</tbody>
</table>

### Code Example

In the following example, an application prepares an SQL statement to insert data into the ORDERS table. For each parameter in the statement, the application calls SQLBindParameter to specify the ODBC C data type and the SQL data type of the parameter, and to bind a buffer to each parameter.
For each row of data, the application assigns data values to each parameter and calls **SQLExecute** to execute the statement.

The following sample assumes that you have an ODBC data source on your computer called Northwind that is associated with the Northwind database.

For more code examples, see **SQLBulkOperations Function**, **SQLProcedures Function**, **SQLPutData Function**, and **SQLSetPos Function**.

```cpp
// SQLBindParameter Function.cpp
// compile with: ODBC32.lib
#include <windows.h>
#include <sqltypes.h>
#include <sqlext.h>
#define EMPLOYEE_ID_LEN 10

SQLHENV henv = NULL;
SQLHDBC hdbc = NULL;
SQLRETURN retcode;
SQLHSTMT hstmt = NULL;
SQLSMALLINT sCustID;

SQLCHAR szEmployeeID[EMPLOYEE_ID_LEN];
SQL_DATE_STRUCT dsOrderDate;
SQLINTEGER cbCustID = 0, cbOrderDate = 0, cbEmployeeID = SQL_NTS;

int main()
{
    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)SQL_OV_ODBC3, 0);
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
    retcode = SQLConnect(hdbc, (SQLCHAR*) "Northwind", SQL_NTS, (SQLCHAR*) NULL, 0, NULL, SQL_NTS);
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    retcode = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, EMPLOYEE_ID_LEN, 0, szEmployeeID, 0, &cbEmployeeID);
    retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_SSHORT, SQL_INTEGER, 0, 0, &sCustID, 0, &cbCustID);
    retcode = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_TYPE_DATE, SQL_TIMESTAMP, sizeof(dsOrderDate), 0, &dsOrderDate, 0, &cbOrderDate);
    retcode = SQLPrepare(hstmt, (SQLCHAR*)"INSERT INTO Orders(CustomerID, EmployeeID, OrderDate) VALUES (?, ?, ?)", SQL_NTS);
    strcpy_s((char*)szEmployeeID, _countof(szEmployeeID), "BERGS");
    sCustID = 5;
    dsOrderDate.year = 2006;
    dsOrderDate.month = 3;
    dsOrderDate.day = 17;
    retcode = SQLExecute(hstmt);
}
```
In the following example, an application executes a SQL Server stored procedure using a named parameter.

```c
#include <windows.h>
#include <sqltypes.h>
#include <sqlext.h>

SQLHDESC hIpd = NULL;
SQLHENV henv = NULL;
SQLHDBC hdbc = NULL;
SQLRETURN retcode;
SQLHSTMT hstmt = NULL;
SQLCHAR szQuote[50] = "100084";
SQLINTEGER cbValue = SQL_NTS;

int main()
{
    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)SQL_OV_ODBC3, 0);
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
    retcode = SQLConnect(hdbc, (SQLCHAR*) "Northwind", SQL_NTS, (SQLCHAR*) NULL, 0, NULL,
                        SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt));
    retcode = SQLPrepare(hstmt, (SQLCHAR*)"{call SQLBindParameter(?)}", SQL_NTS);
    retcode = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 50, 0, sz
                               retcode = SQLGetStmtAttr(hstmt, SQL_ATTR_IMP_PARAM_DESC, &hIpd, 0, 0);
    retcode = SQLSetDescField(hIpd, 1, SQL_DESC_NAME, "@quote", SQL_NTS);
    retcode = SQLExecute(hstmt);
}
```

### Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returning information about a parameter in a statement</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
</tr>
<tr>
<td>SQLDescribeParam Function</td>
</tr>
<tr>
<td>SQLExecDirect Function</td>
</tr>
</tbody>
</table>
Executing a prepared SQL statement | SQLExecute Function
---|---
Releasing parameter buffers on the statement | SQLFreeStmt Function
Returning the number of statement parameters | SQLNumParams Function
Returning the next parameter to send data for | SQLParamData Function
Specifying multiple parameter values | SQLParamOptions Function
Sending parameter data at execution time | SQLPutData Function

See Also
- ODBC API Reference
- ODBC Header Files
- Retrieving Output Parameters Using SQLGetData

SQLBrowseConnect Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ODBC

Summary
SQLBrowseConnect supports an iterative method of discovering and enumerating the attributes and attribute values required to connect to a data source. Each call to SQLBrowseConnect returns successive levels of attributes and attribute values. When all levels have been enumerated, a connection to the data source is completed and a complete connection string is returned by SQLBrowseConnect. A return code of SQL_SUCCESS or SQL_SUCCESS_WITH_INFO indicates that all connection information has been specified and the application is now connected to the data source.

Syntax

```c
SQLRETURN SQLBrowseConnect(
    SQLHDBC    ConnectionHandle,
    SQLCHAR *  InConnectionString,
    SQLSMALLINT StringLength1,
    SQLCHAR *  OutConnectionString,
    SQLSMALLINT BufferLength,
    SQLSMALLINT * StringLength2Ptr);
```
Arguments

ConnectionHandle
[Input] Connection handle.

InConnectionString
[Input] Browse request connection string (see "InConnectionString Argument" in "Comments").

StringLength1
[Input] Length of *InConnectionString in characters.

OutConnectionString
[Output] Pointer to a character buffer in which to return the browse result connection string (see "OutConnectionString Argument" in "Comments").

If OutConnectionString is NULL, StringLength2Ptr will still return the total number of characters (excluding the null-termination character for character data) available to return in the buffer pointed to by OutConnectionString.

BufferLength
[Input] Length, in characters, of the *OutConnectionString buffer.

StringLength2Ptr
[Output] The total number of characters (excluding null-termination) available to return in *OutConnectionString. If the number of characters available to return is greater than or equal to BufferLength, the connection string in *OutConnectionString is truncated to BufferLength minus the length of a null-termination character.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_STILL_EXECUTING.

Diagnostics

When SQLBrowseConnect returns SQL_ERROR, SQL_SUCCESS_WITH_INFO, or SQL_NEED_DATA, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of ConnectionHandle. The following table lists the SQLSTATE values commonly returned by SQLBrowseConnect and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right</td>
<td>The buffer *OutConnectionString was not large enough to</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>truncated</td>
<td>return the entire browse result connection string, so the string was truncated. The buffer *StringLength2Ptr contains the length of the untruncated browse result connection string. (Function returns SQL_NEED_DATA.)</td>
<td></td>
</tr>
<tr>
<td>01S00</td>
<td>Invalid connection string attribute</td>
<td>An invalid attribute keyword was specified in the browse request connection string (InConnectionString). (Function returns SQL_NEED_DATA.) An attribute keyword was specified in the browse request connection string (InConnectionString) that does not apply to the current connection level. (Function returns SQL_NEED_DATA.)</td>
</tr>
<tr>
<td>01S02</td>
<td>Value changed</td>
<td>The driver did not support the specified value of the ValuePtr argument in SQLSetConnectAttr and substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08001</td>
<td>Client unable to establish connection</td>
<td>The driver was unable to establish a connection with the data source.</td>
</tr>
<tr>
<td>08002</td>
<td>Connection name in use</td>
<td>(DM) The specified connection had already been used to establish a connection with a data source, and the connection was open.</td>
</tr>
<tr>
<td>08004</td>
<td>Server rejected the connection</td>
<td>The data source rejected the establishment of the connection for implementation-defined reasons.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was attempting to connect failed before the function completed processing.</td>
</tr>
<tr>
<td>28000</td>
<td>Invalid authorization specification</td>
<td>Either the user identifier or the authorization string or both, as specified in the browse request connection string (InConnectionString), violated restrictions defined by the data source.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>(DM) The Driver Manager was unable to allocate memory required to support execution or completion of the function. The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>An asynchronous operation was canceled by calling SQL CancelHandle Function. Then, the original function was called again on the ConnectionHandle. An operation was canceled by calling SQL CancelHandle on the ConnectionHandle from a different thread in a multithread application.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Message</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function (not this one) was called for the <code>ConnectionHandle</code> and was still executing when this function was called.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value specified for argument <code>StringLength1</code> was less than 0 and was not equal to SQL_NTS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The value specified for argument <code>BufferLength</code> was less than 0.</td>
</tr>
<tr>
<td>HY114</td>
<td>Driver does not support connection level asynchronous function execution</td>
<td>(DM) The application enabled the asynchronous operation on the connection handle before making the connection. However, the driver does not support asynchronous operation on connection handle.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The login timeout period expired before the connection to the data source completed. The timeout period is set through <code>SQLSetConnectAttr</code>, SQL_ATTR_LOGIN_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver corresponding to the specified data source name does not support the function.</td>
</tr>
<tr>
<td>IM002</td>
<td>Data source not found and no default driver specified</td>
<td>(DM) The data source name specified in the browse request connection string (<code>InConnectionString</code>) was not found in the system information, nor was there a default driver specification.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) ODBC data source and default driver information could not be found in the system information.</td>
</tr>
<tr>
<td>IM003</td>
<td>Specified driver could not be loaded</td>
<td>(DM) The driver listed in the data source specification in the system information or specified by the DRIVER keyword was not found or could not be loaded for some other reason.</td>
</tr>
<tr>
<td>IM004</td>
<td>Driver's <code>SQLAllocHandle</code> on SQL_HANDLE_ENV failed</td>
<td>(DM) During <code>SQLBrowseConnect</code>, the Driver Manager called the driver's <code>SQLAllocHandle</code> function with a <code>HandleType</code> of SQL_HANDLE_ENV and the driver returned an error.</td>
</tr>
<tr>
<td>IM005</td>
<td>Driver's <code>SQLAllocHandle</code> on SQL_HANDLE_DBC failed</td>
<td>(DM) During <code>SQLBrowseConnect</code>, the Driver Manager called the driver's <code>SQLAllocHandle</code> function with a <code>HandleType</code> of SQL_HANDLE_DBC and the driver returned an error.</td>
</tr>
</tbody>
</table>
**IM006** | Driver's `SQLSetConnectAttr` failed | (DM) During `SQLBrowseConnect`, the Driver Manager called the driver's `SQLSetConnectAttr` function and the driver returned an error. 

**IM009** | Unable to load translation DLL | The driver was unable to load the translation DLL that was specified for the data source or for the connection. 

**IM010** | Data source name too long | (DM) The attribute value for the DSN keyword was longer than `SQL_MAX_DSN_LENGTH` characters. 

**IM011** | Driver name too long | (DM) The attribute value for the DRIVER keyword was longer than 255 characters. 

**IM012** | DRIVER keyword syntax error | (DM) The keyword-value pair for the DRIVER keyword contained a syntax error. 

**IM014** | The specified DSN contains an architecture mismatch between the Driver and Application | (DM) 32-bit application uses a DSN connecting to a 64-bit driver; or vice versa. 

**IM017** | Polling is disabled in asynchronous notification mode | Whenever the notification model is used, polling is disabled. 

**IM018** | `SQLCompleteAsync` has not been called to complete the previous asynchronous operation on this handle. | If the previous function call on the handle returns `SQL_STILL_EXECUTING` and if notification mode is enabled, `SQLCompleteAsync` must be called on the handle to do post-processing and complete the operation. 

**S1118** | Driver does not support asynchronous notification | When the driver does not support asynchronous notification, you cannot set `SQL_ATTR_ASYNC_DBC_EVENT` or `SQL_ATTR_ASYNC_DBC_RETCODE_PTR`. 

---

**InConnectionString Argument**

A browse request connection string has the following syntax:

```
connection-string ::= attribute[;] | attribute; connection-stringattribute ::= attribute-keyword=attribute-value | DRIVER=[{}attribute-value{}]attribute-keyword ::= DSN | UID | PWD | driver-defined-attribute-keywordattribute-value ::= character-stringdriver-defined-attribute-keyword ::= identifier
```

where `character-string` has zero or more characters; `identifier` has one or more characters; `attribute-keyword` is not case-sensitive; `attribute-value` may be case-sensitive; and the value of the `DSN` keyword does not consist solely of blanks. Because of connection string and initialization file grammar, keywords and attribute values that contain the characters `{}`(),;?*!@ should be avoided. Because of the grammar in the system information, keywords and data source names cannot contain the backslash (\) character. For an ODBC 2.x driver, braces are required around the attribute value for the
If any keywords are repeated in the browse request connection string, the driver uses the value associated with the first occurrence of the keyword. If the **DSN** and **DRIVER** keywords are included in the same browse request connection string, the Driver Manager and driver use whichever keyword appears first.

For information about how an application chooses a data source or driver, see Choosing a Data Source or Driver.

**OutConnectionString Argument**

The browse result connection string is a list of connection attributes. A connection attribute consists of an attribute keyword and a corresponding attribute value. The browse result connection string has the following syntax:

```
connection-string ::= attribute[;] | attribute; connection-stringattribute ::= [*]attribute-keyword=attribute-valueattribute-keyword ::= ODBC-attribute-keyword | driver-defined-attribute-keywordODBC-attribute-keyword = {UID | PWD};[;localized-identifier]driver-defined-attribute-keyword ::= identifier;localized-identifierattribute-value ::= {attribute-value-list} | ? (The braces are literal; they are returned by the driver.)attribute-value-list ::= character-string [:localized-character string], attribute-value-list
```

where **character-string** and **localized-character string** have zero or more characters; **identifier** and **localized-identifier** have one or more characters; **attribute-keyword** is not case-sensitive; and **attribute-value** may be case-sensitive. Because of connection string and initialization file grammar, keywords, localized identifiers, and attribute values that contain the characters `[]{}(),;?*=!@` should be avoided. Because of the grammar in the system information, keywords and data source names cannot contain the backslash (`\`) character.

The browse result connection string syntax is used according to the following semantic rules:

- If an asterisk (*) precedes an **attribute-keyword**, the attribute is optional and can be omitted in the next call to **SQLBrowseConnect**.
- The attribute keywords **UID** and **PWD** have the same meaning as defined in **SQLDriverConnect**.
- A **driver-defined-attribute-keyword** names the kind of attribute for which an attribute value may be supplied. For example, it might be **SERVER**, **DATABASE**, **HOST**, or **DBMS**.
- **ODBC-attribute-keywords** and **driver-defined-attribute-keywords** include a localized or user-friendly version of the keyword. This might be used by applications as a label in a dialog box. However, **UID**, **PWD**, or the **identifier** alone must be used when passing a browse request string to the driver.
- The `{attribute-value-list}` is an enumeration of actual values valid for the corresponding **attribute-keyword**. Note that the braces `{}` do not indicate a list of choices; they are returned by the driver. For example, it might be a list of server names or a list of database names.
- If the **attribute-value** is a single question mark (?), a single value corresponds to the **attribute-keyword**. For example, UID=JohnS; PWD=Sesame.
- Each call to **SQLBrowseConnect** returns only the information required to satisfy the next level of the connection process. The driver associates state information with the connection handle so
Using SQLBrowseConnect

SQLBrowseConnect requires an allocated connection. The Driver Manager loads the driver that was specified in or that corresponds to the data source name specified in the initial browse request connection string; for information about when this occurs, see the "Comments" section in SQLConnect Function. The driver may establish a connection with the data source during the browsing process. If SQLBrowseConnect returns SQL_ERROR, outstanding connections are terminated and the connection is returned to an unconnected state.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
</table>

SQLBrowseConnect does not support connection pooling. If SQLBrowseConnect is called while connection pooling is enabled, SQLSTATE HY000 (General error) will be returned.

When SQLBrowseConnect is called for the first time on a connection, the browse request connection string must contain the DSN keyword or the DRIVER keyword. If the browse request connection string contains the DSN keyword, the Driver Manager locates a corresponding data source specification in the system information:

- If the Driver Manager finds the corresponding data source specification, it loads the associated driver DLL; the driver can retrieve information about the data source from the system information.
- If the Driver Manager cannot find the corresponding data source specification, it locates the default data source specification and loads the associated driver DLL; the driver can retrieve information about the default data source from the system information. "DEFAULT" is passed to the driver for the DSN.
- If the Driver Manager cannot find the corresponding data source specification and there is no default data source specification, it returns SQL_ERROR with SQLSTATE IM002 (Data source not found and no default driver specified).

If the browse request connection string contains the DRIVER keyword, the Driver Manager loads the specified driver; it does not attempt to locate a data source in the system information. Because the DRIVER keyword does not use information from the system information, the driver must define enough keywords so that a driver can connect to a data source using only the information in the browse request connection strings.

On each call to SQLBrowseConnect, the application specifies the connection attribute values in the browse request connection string. The driver returns successive levels of attributes and attribute values in the browse result connection string; it returns SQL_NEED_DATA as long as there are connection attributes that have not yet been enumerated in the browse request connection string. The application uses the contents of the browse result connection string to build the browse request connection string for the next call to SQLBrowseConnect. All mandatory attributes (those not preceded by an asterisk in the OutConnectionString argument) must be included in the next call to SQLBrowseConnect. Note that the application cannot use the contents of previous browse result connection strings when building the current browse request connection string; that is, it cannot specify different values for attributes set in previous levels.

When all levels of connection and their associated attributes have been enumerated, the driver returns
SQL_SUCCESS, the connection to the data source is complete, and a complete connection string is returned to the application. The connection string is suitable to use, in conjunction with SQLDriverConnect, with the SQL_DRIVER_NOPROMPT option to establish another connection. The complete connection string cannot be used in another call to SQLBrowseConnect, however; if SQLBrowseConnect were called again, the entire sequence of calls would have to be repeated.

SQLBrowseConnect also returns SQL_NEED_DATA if there are recoverable, nonfatal errors during the browse process; for example, an invalid password or attribute keyword supplied by the application. When SQL_NEED_DATA is returned and the browse result connection string is unchanged, an error has occurred and the application can call SQLGetDiagRec to return the SQLSTATE for browse-time errors. This permits the application to correct the attribute and continue the browse.

An application can terminate the browse process at any time by calling SQLDisconnect. The driver will terminate any outstanding connections and return the connection to an unconnected state.

If asynchronous operations are enabled on the connection handle, SQLBrowseConnect might also return SQL_STILL_EXECUTING. When it returns SQL_NEED_DATA, an application must use SQLDisconnect to cancel the browse process. If SQLBrowseConnect returns SQL_STILL_EXECUTING, an application should use SQLCancelHandle to cancel the operation in progress. Calling SQLCancelHandle after the function returns SQL_NEED_DATA has no effect.

For more information, see Connecting with SQLBrowseConnect.

If a driver supports SQLBrowseConnect, the driver keyword section in the system information for the driver must contain the ConnectFunctions keyword with the third character set to "Y."

**Code Example**

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you are connecting to a data source provider that supports Windows authentication, you should specify Trusted_Connection=yes instead of user ID and password information in the connection string.</td>
</tr>
</tbody>
</table>

In the following example, an application calls SQLBrowseConnect repeatedly. Each time SQLBrowseConnect returns SQL_NEED_DATA, it passes back information about the data it needs in *OutConnectionString. The application passes OutConnectionString to its routine GetUserInput (not shown). GetUserInput parses the information, builds and displays a dialog box, and returns the information entered by the user in *InConnectionString. The application passes the user's information to the driver in the next call to SQLBrowseConnect. After the application has provided all necessary information for the driver to connect to the data source, SQLBrowseConnect returns SQL_SUCCESS and the application proceeds.

For a more detailed example of connecting to a SQL Server driver by calling SQLBrowseConnect, see SQL Server Browsing Example.

For example, to connect to the data source Sales, the following actions might occur. First, the application passes the following string to SQLBrowseConnect:

```
"DSN=Sales"
```

The Driver Manager loads the driver associated with the data source Sales. It then calls the driver's
**SQLBrowseConnect** function with the same arguments it received from the application. The driver returns the following string in *OutConnectionString*:

```
"HOST:Server={red,blue,green};UID:ID=?;PWD:Password=?"
```

The application passes this string to its **GetUserInput** routine, which builds a dialog box that asks the user to select the red, blue, or green server and to enter a user ID and password. The routine passes the following user-specified information back in *InConnectionString*, which the application passes to **SQLBrowseConnect**:

```
"HOST=red;UID=Smith;PWD=Sesame"
```

**SQLBrowseConnect** uses this information to connect to the red server as Smith with the password Sesame, and then returns the following string in *OutConnectionString*:

```
"*DATABASE:Database={SalesEmployees,SalesGoals,SalesOrders}"}
```

The application passes this string to its **GetUserInput** routine, which builds a dialog box that asks the user to select a database. The user selects empdata and the application calls **SQLBrowseConnect** a final time with this string:

```
"DATABASE=SalesOrders"
```

This is the final piece of information the driver needs to connect to the data source; **SQLBrowseConnect** returns SQL_SUCCESS, and *OutConnectionString* contains the completed connection string:

```
// SQLBrowseConnect_Function.cpp
// compile with: odbc32.lib
#include <windows.h>
#include <sqltypes.h>
#include <sqlext.h>

#define BRWS_LEN 100
SQLHENV henv;
SQLHDBC hdbc;
SQLHSTMT hstmt;
SQLRETURN retcode;
SQLCHAR szConnStrIn[BRWS_LEN], szConnStrOut[BRWS_LEN];
SQLSMALLINT cbConnStrOut;

void GetUserInput(SQLCHAR *szConnStrOut, SQLCHAR *szConnStrIn) {}

int main()
{
    // Allocate the environment handle.
```c
retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

    // Set the version environment attribute.
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)SQL_OV_ODBC3, 0)
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

        // Allocate the connection handle.
        retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
        if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
            // Call SQLBrowseConnect until it returns a value other than SQL_NEED_DATA
            // (pass data source name the first time). If SQL_NEED_DATA is returned, ca
            // (not shown) to build a dialog from the values in szConnStrOut. The user-
            // are returned in szConnStrIn, which is passed in the next call to SQLBrows
            strcpy_s((char*)szConnStrIn, _countof(szConnStrIn), "DSN=Sales");
            do {
                retcode = SQLBrowseConnect(hdbc, szConnStrIn, SQL_NTS,
                szConnStrOut, BRWS_LEN, &cbConnStrOut);
                if (retcode == SQL_NEED_DATA)
                    GetUserInput(szConnStrOut, szConnStrIn);
            } while (retcode == SQL_NEED_DATA);

            if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){

                // Allocate the statement handle.
                retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

                if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
                    // Process data after successful connection
                    SQLDisconnect(hdbc);
            }
        }
    }
    SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
}
}
SQLFreeHandle(SQL_HANDLE_ENV, henv);
```
SQLBulkOperations Function

Conformance
Version Introduced: ODBC 3.0 Standards Compliance: ODBC

Summary
SQLBulkOperations performs bulk insertions and bulk bookmark operations, including update, delete, and fetch by bookmark.

Syntax

```
SQLRETURN SQLBulkOperations(
    SQLHSTMT StatementHandle,
    SQLUSMALLINT Operation);
```

Arguments

*StatementHandle*
[Input] Statement handle.

*Operation*
[Input] Operation to perform:

- SQL_ADD
- SQL_UPDATE_BY_BOOKMARK
- SQL_DELETE_BY_BOOKMARK
- SQL_FETCH_BY_BOOKMARK

For more information, see "Comments."

Returns
When SQLBulkOperations returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values typically returned by SQLBulkOperations and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

For all those SQLSTATEs that can return SQL_SUCCESS_WITH_INFO or SQL_ERROR (except 01xxx SQLSTATEs), SQL_SUCCESS_WITH_INFO is returned if an error occurs on one or more, but not all, rows of a multirow operation, and SQL_ERROR is returned if an error occurs on a single-row operation.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data right truncation</td>
<td>The Operation argument was SQL_FETCH_BY_BOOKMARK, and string or binary data returned for a column or columns with a data type of SQL_C_CHAR or SQL_C_BINARY resulted in the truncation of nonblank character or non-NULL binary data.</td>
</tr>
<tr>
<td>01S01</td>
<td>Error in row</td>
<td>The Operation argument was SQL_ADD, and an error occurred in one or more rows while performing the operation but at least one row was successfully added. (Function returns SQL_SUCCESS_WITH_INFO.) (This error is raised only when an application is working with an ODBC 2.x driver.)</td>
</tr>
<tr>
<td>01S07</td>
<td>Fractional truncation</td>
<td>The Operation argument was SQL_FETCH_BY_BOOKMARK, the data type of the application buffer was not SQL_C_CHAR or SQL_C_BINARY, and the data returned to application buffers for one or more columns was truncated. (For numeric C data types, the fractional part of the number was truncated. For time, timestamp, and interval C data types that contain a time component, the fractional portion of the time was truncated.) (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>The Operation argument was SQL_FETCH_BY_BOOKMARK, and the data value of a column in the result set could not be converted to the data type specified by the TargetType argument in the call to SQLBindCol. The Operation argument was SQL_UPDATE_BY_BOOKMARK</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description and Details</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------------------------</td>
<td></td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index. The argument <code>Operation</code> was SQL_ADD, and a column was bound with a column number greater than the number of columns in the result set.</td>
<td></td>
</tr>
<tr>
<td>21S02</td>
<td>Degree of derived table does not match column list. The argument <code>Operation</code> was SQL_UPDATE_BY_BOOKMARK; and no columns were updatable because all columns were either unbound or read-only, or the value in the bound length/indicator buffer was SQL_COLUMN_IGNORE.</td>
<td></td>
</tr>
<tr>
<td>22001</td>
<td>String data right truncation. The assignment of a character or binary value to a column in the result set resulted in the truncation of nonblank (for characters) or non-null (for binary) characters or bytes.</td>
<td></td>
</tr>
<tr>
<td>22003</td>
<td>Numeric value out of range. The <code>Operation</code> argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of a numeric value to a column in the result set caused the whole (as opposed to fractional) part of the number to be truncated. The argument <code>Operation</code> was SQL_FETCH_BY_BOOKMARK, and returning the numeric value for one or more bound columns would have caused a loss of significant digits.</td>
<td></td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format. The <code>Operation</code> argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of a date or timestamp value to a column in the result set caused the year, month, or day field to be out of range. The argument <code>Operation</code> was SQL_FETCH_BY_BOOKMARK, and returning the date or timestamp value for one or more bound columns would have caused the year, month, or day field to be out of range.</td>
<td></td>
</tr>
<tr>
<td>22008</td>
<td>Date/time field overflow. The <code>Operation</code> argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the performance of datetime arithmetic on data being sent to a column in the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result falling outside the permissible range of values for the field or being invalid based on the Gregorian calendar's natural rules for datetimes. The <code>Operation</code> argument was SQL_FETCH_BY_BOOKMARK, and the performance of datetime arithmetic on data being retrieved from the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result falling outside the permissible range of values for the field or being invalid based on the Gregorian calendar's natural rules for datetimes.</td>
<td></td>
</tr>
</tbody>
</table>
| 22015      | Interval field overflow. The `Operation` argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of an exact numeric or interval C type to an interval SQL data type
caused a loss of significant digits.

The *Operation* argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK; when assigning to an interval SQL type, there was no representation of the value of the C type in the interval SQL type.

The *Operation* argument was SQL_FETCH_BY_BOOKMARK, and assigning from an exact numeric or interval SQL type to an interval C type caused a loss of significant digits in the leading field.

The *Operation* argument was SQL_FETCH_BY_BOOKMARK; when assigning to an interval C type, there was no representation of the value of the SQL type in the interval C type.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>22018</td>
<td>Invalid character value for cast specification</td>
<td>The <em>Operation</em> argument was SQL_FETCH_BY_BOOKMARK; the C type was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The argument <em>Operation</em> was SQL_ADD or SQL_UPDATE_BY_BOOKMARK; the SQL type was an exact or approximate numeric, a datetime, or an interval data type; the C type was SQL_C_CHAR; and the value in the column was not a valid literal of the bound SQL type.</td>
</tr>
<tr>
<td>23000</td>
<td>Integrity constraint violation</td>
<td>The <em>Operation</em> argument was SQL_ADD, SQL_DELETE_BY_BOOKMARK, or SQL_UPDATE_BY_BOOKMARK, and an integrity constraint was violated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <em>Operation</em> argument was SQL_ADD, and a column that was not bound is defined as NOT NULL and has no default.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <em>Operation</em> argument was SQL_ADD, the length specified in the bound StrLen_or_IndPtr buffer was SQL_COLUMN_IGNORE, and the column did not have a default value.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>The <em>StatementHandle</em> was in an executed state, but no result set was associated with the <em>StatementHandle</em>.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back because of a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>42000</td>
<td>Syntax error or access violation</td>
<td>The driver was unable to lock the row as needed to perform the operation requested in the <em>Operation</em> argument.</td>
</tr>
<tr>
<td>44000</td>
<td>WITH CHECK</td>
<td>The <em>Operation</em> argument was SQL_ADD or...</td>
</tr>
<tr>
<td>ERROR</td>
<td>DESCRIPTION</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>OPTION violation</td>
<td>SQL_UPDATE_BY_BOOKMARK, and the insert or update was performed on a viewed table (or a table derived from the viewed table) that was created by specifying <strong>WITH CHECK OPTION</strong>, in such a way that one or more rows affected by the insert or update will no longer be present in the viewed table.</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the <em>MessageText</em> buffer describes the error and its cause.</td>
<td></td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
<td></td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Asynchronous processing was enabled for the StatementHandle. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle. Then the function was called again on the StatementHandle.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle from a different thread in a multithread application.</td>
<td></td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLBulkOperations function was called.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(DM) The specified StatementHandle was not in an executed state. The function was called without first calling SQLExecDirect, SQLExecute, or a catalog function.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(DM) The driver was an ODBC 2.x driver, and SQLBulkOperations was called for a StatementHandle before SQLFetchScroll or SQLFetch was called.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(DM) SQLBulkOperations was called after SQLExtendedFetch was called on the StatementHandle.</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY011</td>
<td>Attribute cannot be set now</td>
<td>(DM) The driver was an ODBC 2.x driver, and the <code>SQL_ATTR_ROW_STATUS_PTR</code> statement attribute was set between calls to <code>SQLFetch</code> or <code>SQLFetchScroll</code> and <code>SQLBulkOperations</code>.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>The <code>Operation</code> argument was <code>SQL_ADD</code> or <code>SQL_UPDATE_BY_BOOKMARK</code>; a data value was not a null pointer; the C data type was <code>SQL_C_BINARY</code> or <code>SQL_C_CHAR</code>; and the column length value was less than 0, but not equal to <code>SQL_DATA_AT_EXEC</code>, <code>SQL_COLUMN_IGNORE</code>, <code>SQL_NTS</code>, or <code>SQL_NULL_DATA</code>, or less than or equal to <code>SQL_LEN_DATA_AT_EXEC_OFFSET</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value in a length/indicator buffer was <code>SQL_DATA_AT_EXEC</code>; the SQL type was either <code>SQL_LONGVARCHAR</code>, <code>SQL_LONGVARBINARY</code>, or a long data source-specific data type; and the <code>SQL_NEED_LONG_DATA_LEN</code> information type in <code>SQLGetInfo</code> was &quot;Y&quot;.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>Operation</code> argument was <code>SQL_ADD</code>, the <code>SQL_ATTR_USE_BOOKMARK</code> statement attribute was set to <code>SQL_UB_VARIABLE</code>, and column 0 was bound to a buffer whose length was not equal to the maximum length for the bookmark for this result set. (This length is available in the <code>SQL_DESC_OCTET_LENGTH</code> field of the IRD and can be obtained by calling <code>SQLDescribeCol</code>, <code>SQLColAttribute</code>, or <code>SQLGetDescField</code>.)</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute identifier</td>
<td>(DM) The value specified for the <code>Operation</code> argument was invalid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>Operation</code> argument was <code>SQL_ADD</code>, <code>SQL_UPDATE_BY_BOOKMARK</code>, or <code>SQL_DELETE_BY_BOOKMARK</code>, and the <code>SQL_ATTR_CONCURRENCY</code> statement attribute was set to <code>SQL_CONCUR_READ_ONLY</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>Operation</code> argument was <code>SQL_DELETE_BY_BOOKMARK</code>, <code>SQL_FETCH_BY_BOOKMARK</code>, or <code>SQL_UPDATE_BY_BOOKMARK</code>, and the bookmark column was not bound or the <code>SQL_ATTR_USE_BOOKMARKS</code> statement attribute was set to <code>SQL_UB_OFF</code>.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see <code>SQLEndTran Function</code>.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Information</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The driver or data source does not support the operation requested in the <em>Operation</em> argument.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the result set. The timeout period is set through <code>SQLSetStmtAttr</code> with an <em>Attribute</em> argument of <code>SQL_ATTR_QUERY_TIMEOUT</code>.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>, <code>SQL_ATTR_CONNECTION_TIMEOUT</code>.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the <code>StatementHandle</code> does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><code>SQLCompleteAsync</code> has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns <code>SQL_STILL_EXECUTING</code> and if notification mode is enabled, <code>SQLCompleteAsync</code> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

### Comments

**Caution**

For information about what statement states `SQLBulkOperations` can be called in and what it must do for compatibility with ODBC 2.x applications, see the Block Cursors, Scrollable Cursors, and Backward Compatibility section in Appendix G: Driver Guidelines for Backward Compatibility.

An application uses `SQLBulkOperations` to perform the following operations on the base table or view that corresponds to the current query:

- Add new rows.
- Update a set of rows where each row is identified by a bookmark.
- Delete a set of rows where each row is identified by a bookmark.
- Fetch a set of rows where each row is identified by a bookmark.

After a call to `SQLBulkOperations`, the block cursor position is undefined. The application has to call `SQLFetchScroll` to set the cursor position. An application should call `SQLFetchScroll` only with a `FetchOrientation` argument of `SQL_FETCH_FIRST`, `SQL_FETCH_LAST`, `SQL_FETCH_ABSOLUTE`, or `SQL_FETCH_BOOKMARK`. The cursor position is undefined if the application calls `SQLFetch` or
**SQLFetchScroll** with a `FetchOrientation` argument of `SQL_FETCH_PRIOR`, `SQL_FETCH_NEXT`, or `SQL_FETCH_RELATIVE`.

A column can be ignored in bulk operations performed by a call to **SQLBulkOperations** by setting the column length/indicator buffer specified in the call to **SQLBindCol**, to `SQL_COLUMN_IGNORE`.

It is not necessary for the application to set the `SQL_ATTR_ROW_OPERATION_PTR` statement attribute when it calls **SQLBulkOperations** because rows cannot be ignored when performing bulk operations with this function.

The buffer pointed to by the `SQL_ATTR_ROWS_FETCHED_PTR` statement attribute contains the number of rows affected by a call to **SQLBulkOperations**.

When the `Operation` argument is `SQL_ADD` or `SQL_UPDATE_BY_BOOKMARK` and the select-list of the query specification associated with the cursor contains more than one reference to the same column, it is driver-defined whether an error is generated or the driver ignores the duplicated references and performs the requested operations.

For more information about how to use **SQLBulkOperations**, see *Updating Data with SQLBulkOperations*.  

### Performing Bulk Inserts

To insert data with **SQLBulkOperations**, an application performs the following sequence of steps:

1. Executes a query that returns a result set.
2. Sets the `SQL_ATTR_ROW_ARRAY_SIZE` statement attribute to the number of rows that it wants to insert.
3. Calls **SQLBindCol** to bind the data that it wants to insert. The data is bound to an array with a size equal to the value of `SQL_ATTR_ROW_ARRAY_SIZE`.
   
   **Note**
   
   The size of the array pointed to by the `SQL_ATTR_ROW_STATUS_PTR` statement attribute should either be equal to `SQL_ATTR_ROW_ARRAY_SIZE` or `SQL_ATTR_ROW_STATUS_PTR` should be a null pointer.

4. Calls **SQLBulkOperations**(*StatementHandle*, SQL_ADD) to perform the insertion.
5. If the application has set the `SQL_ATTR_ROW_STATUS_PTR` statement attribute, it can inspect this array to see the result of the operation.

If an application binds column 0 before it calls **SQLBulkOperations** with an `Operation` argument of SQL_ADD, the driver will update the bound column 0 buffers with the bookmark values for the newly inserted row. For this to occur, the application must have set the `SQL_ATTR_USE_BOOKMARKS` statement attribute to `SQL_UB_VARIABLE` before executing the statement. (This does not work with an ODBC 2.x driver.)

Long data can be added in parts by **SQLBulkOperations**, by using calls to **SQLParamData** and **SQLPutData**. For more information, see "Providing Long Data for Bulk Inserts and Updates" later in this function reference.
It is not necessary for the application to call `SQLFetch` or `SQLFetchScroll` before it calls `SQLBulkOperations` (except when going against an ODBC 2.x driver; see Backward Compatibility and Standards Compliance).

The behavior is driver-defined if `SQLBulkOperations`, with an `Operation` argument of SQL_ADD, is called on a cursor that contains duplicate columns. The driver can return a driver-defined SQLSTATE, add the data to the first column that appears in the result set, or perform other driver-defined behavior.

Performing Bulk Updates by Using Bookmarks

To perform bulk updates by using bookmarks with `SQLBulkOperations`, an application performs the following steps in sequence:

1. Sets the `SQL_ATTR_USE_BOOKMARKS` statement attribute to `SQL_UB_VARIABLE`.
2. Executes a query that returns a result set.
3. Sets the `SQL_ATTR_ROW_ARRAY_SIZE` statement attribute to the number of rows that it wants to update.
4. Calls `SQLBindCol` to bind the data that it wants to update. The data is bound to an array with a size equal to the value of `SQL_ATTR_ROW_ARRAY_SIZE`. It also calls `SQLBindCol` to bind column 0 (the bookmark column).
5. Copies the bookmarks for rows that it is interested in updating into the array bound to column 0.
6. Updates the data in the bound buffers.

**Note**

The size of the array pointed to by the `SQL_ATTR_ROW_STATUS_PTR` statement attribute should be equal to `SQL_ATTR_ROW_ARRAY_SIZE` or `SQL_ATTR_ROW_STATUS_PTR` should be a null pointer.

7. Calls `SQLBulkOperations(StatementHandle, SQL_UPDATE_BY_BOOKMARK)`.

**Note**

If the application has set the `SQL_ATTR_ROW_STATUS_PTR` statement attribute, it can inspect this array to see the result of the operation.

8. Optionally calls `SQLBulkOperations(StatementHandle, SQL_FETCH_BY_BOOKMARK)` to fetch data into the bound application buffers to verify that the update has occurred.

9. If data has been updated, the driver changes the value in the row status array for the appropriate rows to `SQL_ROW_UPDATED`.

Bulk updates performed by `SQLBulkOperations` can include long data by using calls to `SQLParamData` and `SQLPutData`. For more information, see "Providing Long Data for Bulk Inserts and Updates" later in this function reference.

If bookmarks persist across cursors, the application does not need to call `SQLFetch` or
**Performing Bulk Fetches Using Bookmarks**

To perform bulk fetches using bookmarks with `SQLBulkOperations`, an application performs the following steps in sequence:

1. Sets the `SQL_ATTR_USE_BOOKMARKS` statement attribute to `SQL_UB_VARIABLE`.
2. Executes a query that returns a result set.
3. Sets the `SQL_ATTR_ROW_ARRAY_SIZE` statement attribute to the number of rows that it wants to fetch.
4. Calls `SQLBindCol` to bind the data that it wants to fetch. The data is bound to an array with a size equal to the value of `SQL_ATTR_ROW_ARRAY_SIZE`. It also calls `SQLBindCol` to bind column 0 (the bookmark column).
5. Copies the bookmarks for rows that it is interested in fetching into the array bound to column 0. (This assumes that the application has already obtained the bookmarks separately.)

**Note**

The size of the array pointed to by the `SQL_ATTR_ROW_STATUS_PTR` statement attribute should be equal to `SQL_ATTR_ROW_ARRAY_SIZE` or `SQL_ATTR_ROW_STATUS_PTR` should be a null pointer.

6. Calls `SQLBulkOperations(StatementHandle, SQL_FETCH_BY_BOOKMARK)`.
7. If the application has set the `SQL_ATTR_ROW_STATUS_PTR` statement attribute, it can inspect this array to see the result of the operation.

If bookmarks persist across cursors, the application does not need to call `SQLFetch` or `SQLFetchScroll` before fetching by bookmarks. It can use bookmarks that it has stored from a previous cursor. If bookmarks do not persist across cursors, the application has to call `SQLFetch` or `SQLFetchScroll` one time to retrieve the bookmarks.

**Performing Bulk Deletes Using Bookmarks**

To perform bulk deletes using bookmarks with `SQLBulkOperations`, an application performs the following steps in sequence:

1. Sets the `SQL_ATTR_USE_BOOKMARKS` statement attribute to `SQL_UB_VARIABLE`. 
2. Executes a query that returns a result set.

3. Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that it wants to delete.

4. Calls SQLBindCol to bind column 0 (the bookmark column).

5. Copies the bookmarks for rows that it is interested in deleting into the array bound to column 0.

   **Note**
   
   The size of the array pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute should be equal to SQL_ATTR_ROW_ARRAY_SIZE or SQL_ATTR_ROW_STATUS_PTR should be a null pointer.

6. Calls SQLBulkOperations(StatementHandle, SQL_DELETE_BY_BOOKMARK).

7. If the application has set the SQL_ATTR_ROW_STATUS_PTR statement attribute, it can inspect this array to see the result of the operation.

If bookmarks persist across cursors, the application does not have to call SQLFetch or SQLFetchScroll before deleting by bookmarks. It can use bookmarks that it has stored from a previous cursor. If bookmarks do not persist across cursors, the application has to call SQLFetch or SQLFetchScroll one time to retrieve the bookmarks.

### Providing Long Data for Bulk Inserts and Updates

Long data can be provided for bulk inserts and updates performed by calls to SQLBulkOperations. To insert or update long data, an application performs the following steps in addition to the steps described in the "Performing Bulk Inserts" and "Performing Bulk Updates Using Bookmarks" sections earlier in this topic.

1. When it binds the data by using SQLBindCol, the application places an application-defined value, such as the column number, in the *TargetValuePtr buffer for data-at-execution columns. The value can be used later to identify the column.

   The application places the result of the SQL_LEN_DATA_AT_EXEC(length) macro in the *StrLen_or_IndPtr buffer. If the SQL data type of the column is SQL_LONGVARBINARY, SQL_LONGVARCHAR, or a long data source–specific data type and the driver returns "Y" for the SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo, length is the number of bytes of data to be sent for the parameter; otherwise, it must be a nonnegative value and is ignored.

2. When SQLBulkOperations is called, if there are data-at-execution columns, the function returns SQL_NEED_DATA and proceeds to step 3, which follows. (If there are no data-at-execution columns, the process is complete.)

3. The application calls SQLParamData to retrieve the address of the *TargetValuePtr buffer for the first data-at-execution column to be processed. SQLParamData returns SQL_NEED_DATA. The application retrieves the application-defined value from the *TargetValuePtr buffer.

   **Note**
   
   Although data-at-execution parameters resemble data-at-execution columns, the value
Data-at-execution columns are columns in a rowset for which data will be sent with SQLPutData when a row is updated or inserted with SQLBulkOperations. They are bound with SQLBindCol. The value returned by SQLParamData is the address of the row in the *TargetValuePtr buffer that is being processed.

4. The application calls SQLPutData one or more times to send data for the column. More than one call is needed if all the data value cannot be returned in the *TargetValuePtr buffer specified in SQLPutData; multiple calls to SQLPutData for the same column are allowed only when sending character C data to a column with a character, binary, or data source–specific data type or when sending binary C data to a column with a character, binary, or data source–specific data type.

5. The application calls SQLParamData again to signal that all data has been sent for the column.

- If there are more data-at-execution columns, SQLParamData returns SQL_NEED_DATA and the address of the TargetValuePtr buffer for the next data-at-execution column to be processed. The application repeats steps 4 and 5.

- If there are no more data-at-execution columns, the process is complete. If the statement was executed successfully, SQLParamData returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO; if the execution failed, it returns SQL_ERROR. At this point, SQLParamData can return any SQLSTATE that can be returned by SQLBulkOperations.

If the operation is canceled or an error occurs in SQLParamData or SQLPutData after SQLBulkOperations returns SQL_NEED_DATA and before data is sent for all data-at-execution columns, the application can call only SQLCancel, SQLGetDiagField, SQLGetDiagRec, SQLGetFunctions, SQLParamData, or SQLPutData for the statement or the connection associated with the statement. If it calls any other function for the statement or the connection associated with the statement, the function returns SQL_ERROR and SQLSTATE HY010 (Function sequence error).

If the application calls SQLCancel while the driver still needs data for data-at-execution columns, the driver cancels the operation. The application can then call SQLBulkOperations again; canceling does not affect the cursor state or the current cursor position.

Row Status Array

The row status array contains status values for each row of data in the rowset after a call to SQLBulkOperations. The driver sets the status values in this array after a call to SQLFetch, SQLFetchScroll, SQLSetPos, or SQLBulkOperations. This array is initially populated by a call to SQLBulkOperations if SQLFetch or SQLFetchScroll has not been called before SQLBulkOperations. This array is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute. The number of elements in the row status arrays must equal the number of rows in the rowset (as defined by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute). For information about this row status array, see SQLFetch.

Code Example

The following example fetches 10 rows of data at a time from the Customers table. It then prompts the user for an action to take. To reduce network traffic, the example buffer updates, deletes, and inserts locally in the bound arrays, but at offsets past the rowset data. When the user chooses to send
updates, deletes, and inserts to the data source, the code sets the binding offset appropriately and calls `SQLBulkOperations`. For simplicity, the user cannot buffer more than 10 updates, deletes, or inserts.

```c
// SQLBulkOperations_Function.cpp
// compile with: ODBC32.lib
#include <windows.h>
#include <sqlext.h>
#include "stdio.h"
#define UPDATE_ROW 100
#define DELETE_ROW 101
#define ADD_ROW 102
#define SEND_TO_DATA_SOURCE 103
#define UPDATE_OFFSET 10
#define INSERT_OFFSET 20
#define DELETE_OFFSET 30

// Define structure for customer data (assume 10 byte maximum bookmark size).
typedef struct tagCustStruct {
    SQLCHAR Bookmark[10];
    SQLINTEGER BookmarkLen;
    SQLINTEGER CustomerID;
    SQLINTEGER CustIDInd;
    SQLCHAR CompanyName[51];
    SQLINTEGER NameLenOrInd;
    SQLCHAR Address[51];
    SQLINTEGER AddressLenOrInd;
    SQLCHAR Phone[11];
    SQLINTEGER PhoneLenOrInd;
} CustStruct;

// Allocate 40 of these structures. Elements 0-9 are for the current rowset, elements 10-19 are for the buffered updates, elements 20-29 are for the buffered inserts, and elements 30-39 are for the buffered deletes.
CustStruct CustArray[40];
SQLUSMALLINT RowStatusArray[10], Action, RowNum, NumUpdates = 0, NumInserts = 0, NumDeletes = 0;
SQLLEN BindOffset = 0;
SQLRETURN retcode;
SQLHENV henv = NULL;
SQLHDBC hdc = NULL;
SQLHSTMT hstmt = NULL;

int main() {
    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)SQL_OV_ODBC3, 0);
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
    retcode = SQLConnect(hdbc, (SQLCHAR*)"Northwind", SQL_NTS, (SQLCHAR*)NULL, 0, NULL, NULL);
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

    // Set the following statement attributes:
    // SQL_ATTR_CURSOR_TYPE: Keyset-driven
    // SQL_ATTR_ROW_BIND_TYPE: Row-wise
```
// SQL_ATTR_ROW_ARRAY_SIZE: 10
// SQL_ATTR_USE_BOOKMARKS: Use variable-length bookmarks
// SQL_ATTR_ROW_STATUS_PTR: Points to RowStatusArray
// SQL_ATTR_ROW_BIND_OFFSET_PTR: Points to BindOffset
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_CURSOR_TYPE, (SQLPOINTER)SQL_CURSOR_KEYSET_D
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE, (SQLPOINTER)sizeof(CustStruct
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)10, 0);
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_USE_BOOKMARKS, (SQLPOINTER)SQL_UB_VARIABLE,
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_OFFSET_PTR, &BindOffset, 0);

// Bind arrays to the bookmark, CustomerID, CompanyName, Address, and Phone columns.
retcode = SQLBindCol(hstmt, 0, SQL_C_VARBOOKMARK, CustArray[0].Bookmark, sizeof(CustArray[0].Bookmark), &CustArray[0].BookmarkLen);
retcode = SQLBindCol(hstmt, 1, SQL_C_ULONG, &CustArray[0].CustomerID, 0, &CustArray[0].CustIDInd);
retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, CustArray[0].CompanyName, sizeof(CustArray[0].CompanyName), &CustArray[0].NameLenOrInd);
retcode = SQLBindCol(hstmt, 3, SQL_C_CHAR, CustArray[0].Address, sizeof(CustArray[0].Address), &CustArray[0].AddressLenOrInd);
retcode = SQLBindCol(hstmt, 4, SQL_C_CHAR, CustArray[0].Phone, sizeof(CustArray[0].Phone), &CustArray[0].PhoneLenOrInd);

// Execute a statement to retrieve rows from the Customers table.
retcode = SQLExecDirect(hstmt, (SQLCHAR*)"SELECT CustomerID, CompanyName, Address, Phone FROM Customers", SQL_NTS);

// Fetch and display the first 10 rows.
retcode = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
// DisplayCustData(CustArray, 10);

// Call GetAction to get an action and a row number from the user.
// while (GetAction(&Action, &RowNum)) {
    Action = SQL_FETCH_NEXT;
    RowNum = 2;
    switch (Action) {
        case SQL_FETCH_NEXT:
        case SQL_FETCH_PRIOR:
        case SQL_FETCH_FIRST:
        case SQL_FETCH_LAST:
        case SQL_FETCH_ABSOLUTE:
            case SQL_FETCH_RELATIVE:
                // Fetch and display the requested data.
                SQLFetchScroll(hstmt, Action, RowNum);
                // DisplayCustData(CustArray, 10);
                break;

        case UPDATE_ROW:
            // Check if we have reached the maximum number of buffered updates.
            if (NumUpdates < 10) {
                // Get the new customer data and place it in the next available element of
                // the buffered updates section of CustArray, copy the bookmark of the row
                // being updated to the same element, and increment the update counter.
                // Checking to see we have not already buffered an update for this
                // row not shown.
                GetNewCustData(CustArray, UPDATE_OFFSET + NumUpdates);
                memcpy(CustArray[UPDATE_OFFSET + NumUpdates].Bookmark,
                       CustArray[RowNum - 1].Bookmark,
                       CustArray[RowNum - 1].BookmarkLen);
                CustArray[UPDATE_OFFSET + NumUpdates].BookmarkLen =
                        CustArray[RowNum - 1].BookmarkLen;
                NumUpdates++;
            } else {
                printf("Buffers full. Send buffered changes to the data source.");
            }
    }
break;

case DELETE_ROW:
    // Check if we have reached the maximum number of buffered deletes.
    if (NumDeletes < 10) {
        // Copy the bookmark of the row being deleted to the next available element
        // of the buffered deletes section of CustArray and increment the delete
        // counter. Checking to see we have not already buffered an update for
        // this row not shown.
        memcpy(CustArray[DELETE_OFFSET + NumDeletes].Bookmark,
               CustArray[RowNum - 1].Bookmark,
               CustArray[RowNum - 1].BookmarkLen);

        CustArray[DELETE_OFFSET + NumDeletes].BookmarkLen =
               CustArray[RowNum - 1].BookmarkLen;

        NumDeletes++;
    } else
        printf("Buffers full. Send buffered changes to the data source.");
    break;

case ADD_ROW:
    // reached maximum number of buffered inserts?
    if (NumInserts < 10) {
        // Get the new customer data and place it in the next available element of
        // the buffered inserts section of CustArray and increment insert counter.
        // GetNewCustData(CustArray, INSERT_OFFSET + NumInserts);
        NumInserts++;
    } else
        printf("Buffers full. Send buffered changes to the data source.");
    break;

case SEND_TO_DATA_SOURCE:
    // If there are any buffered updates, inserts, or deletes, set the array size
    // to that number, set the binding offset to use the data in the buffered
    // update, insert, or delete part of CustArray, and call SQLBulkOperations to
    // do the updates, inserts, or deletes. Because we will never have more than
    // 10 updates, inserts, or deletes, we can use the same row status array.
    if (NumUpdates)
        SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)NumUpdates, 0);
        BindOffset = UPDATE_OFFSET * sizeof(CustStruct);
        SQLBulkOperations(hstmt, SQL_UPDATE_BY_BOOKMARK);
        NumUpdates = 0;
    }

    if (NumInserts)
        SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)NumInserts, 0);
        BindOffset = INSERT_OFFSET * sizeof(CustStruct);
        SQLBulkOperations(hstmt, SQL_ADD);
        NumInserts = 0;
    }

    if (NumDeletes)
        SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)NumDeletes, 0);
        BindOffset = DELETE_OFFSET * sizeof(CustStruct);
        SQLBulkOperations(hstmt, SQL_DELETE_BY_BOOKMARK);
        NumDeletes = 0;
    }

    // If there were any updates, inserts, or deletes, reset the binding offset
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Getting a single field of a descriptor</td>
<td>SQLGetDescField Function</td>
</tr>
<tr>
<td>Getting multiple fields of a descriptor</td>
<td>SQLGetDescRec Function</td>
</tr>
<tr>
<td>Setting a single field of a descriptor</td>
<td>SQLSetDescField Function</td>
</tr>
<tr>
<td>Setting multiple fields of a descriptor</td>
<td>SQLSetDescRec Function</td>
</tr>
<tr>
<td>Positioning the cursor, refreshing data in the rowset, or updating or deleting data in the rowset</td>
<td>SQLSetPos Function</td>
</tr>
<tr>
<td>Setting a statement attribute</td>
<td>SQLSetStmtAttr Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files
SQLCancel Function

Conformance

Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary

SQLCancel cancels the processing on a statement. To cancel processing on a connection or statement, use SQLCancelHandle Function.

Syntax

```c
SQLRETURN SQLCancel(
    SQLHSTMT StatementHandle);
```

Arguments

StatementHandle

[Input] Statement handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLCancel returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLCancel and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the argument *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLCancel function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) Cancel operation failed because an asynchronous operation is in progress on a connection handle that is associated with StatementHandle.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY018</td>
<td>Server declined cancel request</td>
<td>The server declined the cancel request.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td></td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
</tbody>
</table>

**Comments**

SQLCancel can cancel the following types of processing on a statement:

- A function running asynchronously on the statement.
- A function on a statement that needs data.
- A function running on the statement on another thread.

In ODBC 2.x, if an application calls SQLCancel when no processing is being done on the statement, SQLCancel has the same effect as SQLFreeStmt with the SQL_CLOSE option; this behavior is
defined only for completeness and applications should call SQLFreeStmt or SQLCloseCursor to close cursors.

When SQLCancel is called to cancel a function running asynchronously in a statement or a function on a statement that needs data, diagnostic records posted by the function being canceled are cleared, and SQLCancel posts its own diagnostic records; when SQLCancel is called to cancel a function running on a statement on another thread, however, it does not clear the diagnostic records of the being canceled function and does not post its own diagnostic records.

Canceling Asynchronous Processing

After an application calls a function asynchronously, it calls the function repeatedly to determine whether it has finished processing. If the function is still processing, it returns SQL_STILL_EXECUTING. If the function has finished processing, it returns a different code.

After any call to the function that returns SQL_STILL_EXECUTING, an application can call SQLCancel to cancel the function. If the cancel request is successful, the driver returns SQL_SUCCESS. This message does not indicate that the function was actually canceled; it indicates that the cancel request was processed. When or if the function is actually canceled is driver-dependent and data source-dependent. The application must continue to call the original function until the return code is not SQL_STILL_EXECUTING. If the function was successfully canceled, the return code is SQL_ERROR and SQLSTATE HY008 (Operation canceled). If the function completed its normal processing, the return code is SQL_SUCCESS or SQL_SUCCESS_WITH_INFO if the function succeeded or SQL_ERROR and a SQLSTATE other than HY008 (Operation canceled) if the function failed.

Note

In ODBC 3.5, a call to SQLCancel when no processing is being done on the statement is not treated as SQLFreeStmt with the SQL_CLOSE option, but has no effect at all. To close a cursor, an application should call SQLCloseCursor, not SQLCancel.

For more information about asynchronous processing, see Asynchronous Execution.

Canceling Functions that Need Data

After SQLExecute or SQLExecDirect returns SQL_NEED_DATA and before data has been sent for all data-at-execution parameters, an application can call SQLCancel to cancel the statement execution. After the statement has been canceled, the application can call SQLExecute or SQLExecDirect again. For more information, see SQLBindParameter.

After SQLBulkOperations or SQLSetPos returns SQL_NEED_DATA and before data has been sent for all data-at-execution columns, an application can call SQLCancel to cancel the operation. After the operation has been canceled, the application can call SQLBulkOperations or SQLSetPos again; canceling does not affect the cursor state or the current cursor position. For more information, see SQLBulkOperations or SQLSetPos.

Canceling Functions Executing on Another Thread

In a multithread application, the application can cancel a function that is running on another thread. To cancel the function, the application calls SQLCancel with the same statement handle as that used
by the target function, but on a different thread. How the function is canceled depends on the driver and the operating system. As in canceling a function running asynchronously, the return code of the SQLCancel indicates only whether the driver processed the request successfully. Only SQL_SUCCESS or SQL_ERROR can be returned; no diagnostic information is returned. If the original function is canceled, it returns SQL_ERROR and SQLSTATE HY008 (Operation canceled).

If an SQL statement is being executed when SQLCancel is called on another thread to cancel the statement execution, it is possible for the execution to succeed and return SQL_SUCCESS while the cancel is also successful. In this case, the Driver Manager assumes that the cursor opened by the statement execution is closed by the cancel, so the application will not be able to use the cursor.

For more information about threading, see Multithreading.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a parameter</td>
<td>SQLBindParameter Function</td>
</tr>
<tr>
<td>Performing bulk insert or update operations</td>
<td>SQLBulkOperations Function</td>
</tr>
<tr>
<td>Cancels a function running asynchronously on a connection handle, in addition to the functionality of SQLCancel.</td>
<td>SQLCancelHandle Function</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
<tr>
<td>Freeing a statement handle</td>
<td>SQLFreeStmt</td>
</tr>
<tr>
<td>Obtaining a field of a diagnostic record or a field of the diagnostic header</td>
<td>SQLGetDiagField Function</td>
</tr>
<tr>
<td>Obtaining multiple fields of a diagnostic data structure</td>
<td>SQLGetDiagRec Function</td>
</tr>
<tr>
<td>Returning the next parameter to send data for</td>
<td>SQLParamData Function</td>
</tr>
<tr>
<td>Sending parameter data at execution time</td>
<td>SQLPutData Function</td>
</tr>
<tr>
<td>Positioning the cursor in a rowset, refreshing data in the rowset, or updating or deleting data in the result set</td>
<td>SQLSetPos Function</td>
</tr>
</tbody>
</table>
SQLCancelHandle Function

Conformance
Version Introduced: ODBC 3.8
Standards Compliance: None

It is expected that most ODBC 3.8 (and later) drivers will implement this function. If a driver does not, a call to `SQLCancelHandle` with a connection handle in the `Handle` parameter will return SQL_ERROR with a SQLSTATE of IM001 and message "Driver does not support this function" A call to `SQLCancelHandle` with a statement handle as the `Handle` parameter will be mapped to a call to `SQLCancel` by the Driver Manager and can be processed if the driver implements `SQLCancel`. An application can use `SQLGetFunctions` to determine if a driver supports `SQLCancelHandle`.

Summary
`SQLCancelHandle` cancels the processing on a connection or statement. The Driver Manager maps a call to `SQLCancelHandle` to a call to `SQLCancel` when `HandleType` is `SQL_HANDLE_STMT`.

Syntax

```c
SQLRETURN SQLCancelHandle(
    SQLSMALLINT    HandleType,
    SQLHANDLE      Handle);
```

Arguments

`HandleType`
[Input] The type of the handle on which to cancel processing. Valid values are `SQL_HANDLE_DBC` or `SQL_HANDLE_STMT`.

`Handle`
[Input] The handle on which to cancel processing.

If `Handle` is not a valid handle of the type specified by `HandleType`, `SQLCancelHandle` returns `SQL_INVALID_HANDLE`. 
**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When **SQLCancelHandle** returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling **SQLGetDiagRec** with a **HandleType** of SQL_HANDLE_STMT and a statement handle **Handle** or a **HandleType** of SQL_HANDLE_DBC and a connection handle **Handle**.

The following table lists the SQLSTATE values commonly returned by **SQLCancelHandle** and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <strong>SQLGetDiagRec</strong> in the argument <em>MessageText</em> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
</tbody>
</table>
| HY010    | Function sequence error | An asynchronously executing, statement-related function was called for one of the statement handles associated with the **Handle**, and **HandleType** was set to SQL_HANDLE_DBC. The asynchronous function was still executing when **SQLCancelHandle** was called.  

(DM) The **HandleType** argument was SQL_HANDLE_STMT; an asynchronously executing function was called on the associated connection handle; and the function was still executing when this function was called.

(DM) **SQLExecute**, **SQLExecDirect**, or **SQLMoreResults** was called for one of the statement handles associated with the **Handle** and **HandleType** was set to SQL_HANDLE_DBC, and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.

**SQLBrowseConnect** was called for **ConnectionHandle**, and returned SQL_NEED_DATA. This function was called before the browsing process completed.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute/options identifier</td>
<td>HandleType was set to SQL_HANDLE_ENV or SQL_HANDLE_DESC.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the Handle does not support the function.</td>
</tr>
</tbody>
</table>

If SQLCancelHandle is called with HandleType set to SQL_HANDLE_STMT, it can return any SQLSTATE that can be returned by the function SQLCancel.

**Comments**

This function is similar to SQLCancel but may take either a connection or statement handle as a parameter rather than only a statement handle. The Driver Manager maps a call to SQLCancelHandle to a call to SQLCancel when HandleType is SQL_HANDLE_STMT. This allows applications to use SQLCancelHandle to cancel statement operations even if the driver does not implement SQLCancelHandle.

For more information about cancelling a statement operation, see SQLCancel Function.

If there are no operations in progress on Handle the call to SQLCancelHandle has no effect.

SQLCancelHandle on a connection handle can cancel the following types of processing:

- A function running asynchronously on the connection.
- A function running on the connection handle on another thread.

When SQLCancelHandle is called to cancel a function running asynchronously in a connection, diagnostic records posted by SQLCancelHandle are appended to those returned by the operation being canceled; SQLCancelHandle does not return diagnostic records, however, when canceling a function running on a connection on another thread.

Using SQLCancelHandle to cancel SQLEndTran may put the connection in suspended state. For
Canceling Connection–Related Asynchronous Processing

If a function returns SQL_STILL_EXECUTING, an application can call SQLCancelHandle to cancel the operation. If the cancel request is successful, SQLCancelHandle returns SQL_SUCCESS. This does not mean that the original function was canceled; it indicates that the cancel request was processed. The driver and data source determine when or if the operation is canceled. The application must continue to call the original function until the return code is not SQL_STILL_EXECUTING. If the original function was canceled, the return code is SQL_ERROR and SQLSTATE HY008 (Operation canceled). If the original function completed its normal processing (was not cancelled), the return code is SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, or SQL_ERROR and a SQLSTATE other than HY008 (Operation canceled), if the original function failed.

Canceling Functions Executing on Another Thread

In a multithread application, the application can cancel an operation that is running on another thread. To cancel the operation, the application calls SQLCancelHandle with the handle used by the function, but on a different thread. The driver and operating system determine how the operation is canceled. The SQLCancelHandle return code indicates whether the driver processed the request, returning either SQL_SUCCESS or SQL_ERROR (no diagnostic information is returned). If processing on the original function is canceled, the original function returns SQL_ERROR and SQLSTATE HY008 (Operation cancelled).

If a function is being executed when SQLCancelHandle is called on another thread to cancel the function, it is possible for the function to succeed and return SQL_SUCCESS before the cancel can take effect. A call to SQLCancelHandle has no effect if the operation completed before SQLCancelHandle was able to cancel the operation.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canceling a function running asynchronously on a statement handle, canceling a function on a statement that needs data, or canceling a function running on a statement on another thread.</td>
<td>SQLCancel Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
SQLCloseCursor Function

Conformance
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary
SQLCloseCursor closes a cursor that has been opened on a statement and discards pending results.

Syntax

```c
SQLRETURN SQLCloseCursor(
    SQLHSTMT StatementHandle);
```

Arguments

StatementHandle
[Input] Statement handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLCloseCursor returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLCloseCursor and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>No cursor was open on the <code>StatementHandle</code>. (This is returned only by an ODBC 3.x driver.)</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle associated with the <code>StatementHandle</code> and was still executing when this function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) An asynchronously executing function was called for the <code>StatementHandle</code> and was still executing when this function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for the <code>StatementHandle</code> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the <code>StatementHandle</code> does not support the function.</td>
</tr>
</tbody>
</table>

**Comments**

`SQLCloseCursor` returns SQLSTATE 24000 (Invalid cursor state) if no cursor is open. Calling
**SQLCloseCursor** is equivalent to calling **SQLFreeStmt** with the SQL_CLOSE option, with the exception that **SQLFreeStmt** with SQL_CLOSE has no effect on the application if no cursor is open on the statement, while **SQLCloseCursor** returns SQLSTATE 24000 (Invalid cursor state).

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>If an ODBC 3.x application working with an ODBC 2.x driver calls <strong>SQLCloseCursor</strong> when no cursor is open, SQLSTATE 24000 (Invalid cursor state) is not returned, because the Driver Manager maps <strong>SQLCloseCursor</strong> to <strong>SQLFreeStmt</strong> with SQL_CLOSE. For more information, see Closing the Cursor.</td>
</tr>
</tbody>
</table>

**Code Example**

See **SQLBrowseConnect Function** and **SQLConnect Function**.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canceling statement processing</td>
<td><strong>SQLCancel Function</strong></td>
</tr>
<tr>
<td>Freeing a handle</td>
<td><strong>SQLFreeHandle Function</strong></td>
</tr>
<tr>
<td>Processing multiple result sets</td>
<td><strong>SQLMoreResults Function</strong></td>
</tr>
</tbody>
</table>

**See Also**

- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft

**SQLColAttribute Function**

**Conformance**

Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

**Summary**

**SQLColAttribute** returns descriptor information for a column in a result set. Descriptor information is returned as a character string, a descriptor-dependent value, or an integer value.
Syntax

```
SQLRETURN SQLColAttribute (
    SQLHSTMT  StatementHandle,
    SQLUSMALLINT  ColumnNumber,
    SQLUSMALLINT  FieldIdentifier,
    SQLPOINTER CharacterAttributePtr,
    SQLSMALLINT  BufferLength,
    SQLSMALLINT  * StringLengthPtr,
    SQLLEN  *  NumericAttributePtr);
```

Arguments

**StatementHandle**

[Input] Statement handle.

**ColumnNumber**

[Input] The number of the record in the IRD from which the field value is to be retrieved. This argument corresponds to the column number of result data, ordered sequentially in increasing column order, starting at 1. Columns can be described in any order.

Column 0 can be specified in this argument, but all values except SQL_DESC_TYPE and SQL_DESC_OCTET_LENGTH will return undefined values.

**FieldIdentifier**

[Input] The descriptor handle. This handle defines which field in the IRD should be queried (for example, SQL_COLUMN_TABLE_NAME).

**CharacterAttributePtr**

[Output] Pointer to a buffer in which to return the value in the FieldIdentifier field of the ColumnNumber row of the IRD, if the field is a character string. Otherwise, the field is unused.

If CharacterAttributePtr is NULL, StringLengthPtr will still return the total number of bytes (excluding the null-termination character for character data) available to return in the buffer pointed to by CharacterAttributePtr.

**BufferLength**

[Input] If FieldIdentifier is an ODBC-defined field and CharacterAttributePtr points to a character string or binary buffer, this argument should be the length of *CharacterAttributePtr. If FieldIdentifier is an ODBC-defined field and *CharacterAttributePtr is an integer, this field is ignored. If the *CharacterAttributePtr is a Unicode string (when calling SQLColAttributeW), the BufferLength argument must be an even number. If FieldIdentifier is a driver-defined field, the application indicates the nature of the field to the Driver Manager by setting the BufferLength argument. BufferLength can have the following values:
• If `CharacterAttributePtr` is a pointer to a pointer, `BufferLength` should have the value `SQL_IS_POINTER`.

• If `CharacterAttributePtr` is a pointer to a character string, the `BufferLength` is the length of the buffer.

• If `CharacterAttributePtr` is a pointer to a binary buffer, the application places the result of the `SQL_LEN_BINARY_ATTR(length)` macro in `BufferLength`. This places a negative value in `BufferLength`.

• If `CharacterAttributePtr` is a pointer to a fixed-length data type, `BufferLength` must be one of the following: `SQL_IS_INTEGER`, `SQL_IS_UNINTEGER`, `SQL_SMALLINT`, or `SQLUSMALLINT`.

`StringLengthPtr`

[Output] Pointer to a buffer in which to return the total number of bytes (excluding the null-termination byte for character data) available to return in `*CharacterAttributePtr`.

For character data, if the number of bytes available to return is greater than or equal to `BufferLength`, the descriptor information in `*CharacterAttributePtr` is truncated to `BufferLength` minus the length of a null-termination character and is null-terminated by the driver.

For all other types of data, the value of `BufferLength` is ignored and the driver assumes the size of `*CharacterAttributePtr` is 32 bits.

`NumericAttributePtr`

[Output] Pointer to an integer buffer in which to return the value in the `FieldIdentifier` field of the `ColumnNumber` row of the IRD, if the field is a numeric descriptor type, such as `SQL_DESC_COLUMN_LENGTH`. Otherwise, the field is unused. Please note that some drivers may only write the lower 32-bit or 16-bit of a buffer and leave the higher-order bit unchanged. Therefore, applications should initialize the value to 0 before calling this function.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When `SQLColAttribute` returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling `SQLGetDiagRec` with a `HandleType` of SQL_HANDLE_STMT and a `Handle` of `StatementHandle`. The following table lists the SQLSTATE values commonly returned by `SQLColAttribute` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The buffer *CharacterAttributePtr was not large enough to return the entire string value, so the string value was truncated. The length of the untruncated string value is returned in *StringLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>07005</td>
<td>Prepared statement not a cursor-specification</td>
<td>The statement associated with the StatementHandle did not return a result set and FieldIdentifier was not SQL_DESC_COUNT. There were no columns to describe.</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>(DM) The value specified for ColumnNumber was equal to 0, and the SQL_ATTR_USE_BOOKMARKS statement attribute was SQL_UB_OFF. The value specified for the argument ColumnNumber was greater than the number of columns in the result set.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagField from the diagnostic data structure describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the StatementHandle. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle. Then the function was called again on the StatementHandle. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when SQLColAttribute was called. (DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. (DM) The function was called prior to calling SQLPrepare, SQLExecDirect, or a catalog function for the StatementHandle. (DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called.</td>
</tr>
</tbody>
</table>
When called after **SQLPrepare** and before **SQLExecute**, **SQLColAttribute** can return any SQLSTATE that can be returned by **SQLPrepare** or **SQLExecute**, depending on when the data source evaluates the SQL statement associated with the **StatementHandle**.

For performance reasons, an application should not call **SQLColAttribute** before executing a statement.
Comments

For information about how applications use the information returned by SQLColAttribute, see Result Set Metadata.

SQLColAttribute returns information either in *NumericAttributePtr or in *CharacterAttributePtr. Integer information is returned in *NumericAttributePtr as a SQLLEN value; all other formats of information are returned in *CharacterAttributePtr. When information is returned in *NumericAttributePtr, the driver ignores CharacterAttributePtr, BufferLength, and StringLengthPtr. When information is returned in *CharacterAttributePtr, the driver ignores NumericAttributePtr.

SQLColAttribute returns values from the descriptor fields of the IRD. The function is called with a statement handle rather than a descriptor handle. The values returned by SQLColAttribute for the FieldIdentifier values listed later in this section can also be retrieved by calling SQLGetDescField with the appropriate IRD handle.

The currently defined descriptor fields, the version of ODBC in which they were introduced, and the arguments in which information is returned for them are shown later in this section; more descriptor types may be defined by drivers to take advantage of different data sources.

An ODBC 3.x driver must return a value for each of the descriptor fields. If a descriptor field does not apply to a driver or data source and unless otherwise stated, the driver returns 0 in *StringLengthPtr or an empty string in *CharacterAttributePtr.

Backward Compatibility

The ODBC 3.x function SQLColAttribute replaces the deprecated ODBC 2.x function SQLColAttributes. When mapping SQLColAttributes to SQLColAttribute (when an ODBC 2.x application is working with an ODBC 3.x driver), or mapping SQLColAttribute to SQLColAttributes (when an ODBC 3.x application is working with an ODBC 2.x driver), the Driver Manager either passes the value of FieldIdentifier through, maps it to a new value, or returns an error, as follows:

Note

The prefix used in FieldIdentifier values in ODBC 3.x has been changed from that used in ODBC 2.x. The new prefix is "SQL_DESC"; the old prefix was "SQL_COLUMN".

- If the #define value of the ODBC 2.x FieldIdentifier is the same as the #define value of the ODBC 3.x FieldIdentifier, the value in the function call is just passed through.

- The #define values of the ODBC 2.x FieldIdentifiers SQL_COLUMN_LENGTH, SQL_COLUMN_PRECISION, and SQL_COLUMN_SCALE are different from the #define values of the ODBC 3.x FieldIdentifiers SQL_DESC_PRECISION, SQL_DESC_SCALE, and SQL_DESC_LENGTH. An ODBC 2.x driver need only support the ODBC 2.x values. An ODBC 3.x driver must support both "SQL_COLUMN" and "SQL_DESC" values for these three FieldIdentifiers. These values are different because precision, scale, and length are defined differently in ODBC 3.x than they were in ODBC 2.x. For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size.

- If the #define value of the ODBC 2.x FieldIdentifier is different from the #define value of the ODBC 3.x FieldIdentifier, as occurs with the COUNT, NAME, and NULLABLE values, the value in the function call is mapped to the corresponding value. For example, SQL_COLUMN_COUNT is mapped to SQL_DESC_COUNT, and SQL_DESC_COUNT is mapped to SQL_COLUMN_COUNT,
depending on the direction of the mapping.

- If FieldIdentifier is a new value in ODBC 3.x, for which there was no corresponding value in ODBC 2.x, it will not be mapped when an ODBC 3.x application uses it in a call to SQLColAttribute in an ODBC 2.x driver, and the call will return SQLSTATE HY091 (Invalid descriptor field identifier).

The following table lists the descriptor types returned by SQLColAttribute. The type for NumericAttributePtr values is SQLLEN.*

<table>
<thead>
<tr>
<th>FieldIdentifier</th>
<th>Information returned in</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_AUTO_UNIQUE_VALUE (ODBC 1.0)</td>
<td>NumericAttributePtr</td>
<td>SQL_TRUE if the column is an autoincrementing column. SQL_FALSE if the column is not an autoincrementing column or is not numeric. This field is valid for numeric data type columns only. An application can insert values into a row containing an autoincrement column, but typically cannot update values in the column. When an insert is made into an autoincrement column, a unique value is inserted into the column at insert time. The increment is not defined, but is data source-specific. An application should not assume that an autoincrement column starts at any particular point or increments by any particular value.</td>
</tr>
<tr>
<td>SQL_DESC_BASE_COLUMN_NAME (ODBC 3.0)</td>
<td>CharacterAttributePtr</td>
<td>The base column name for the result set column. If a base column name does not exist (as in the case of columns that are expressions), then this variable contains an empty string. This information is returned from the SQL_DESC_BASE_COLUMN_NAME record field of the IRD, which is a read-only field.</td>
</tr>
<tr>
<td>SQL_DESC_BASE_TABLE_NAME (ODBC 3.0)</td>
<td>CharacterAttributePtr</td>
<td>The name of the base table that contains the column. If the base table name cannot be defined or is not applicable, then this variable contains an empty string.</td>
</tr>
<tr>
<td>SQL_Desc</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SQL_DESC_BASE_TABLE_NAME (ODBC 1.0)</td>
<td>NumericAttributePtr</td>
<td>This information is returned from the SQL_DESC_BASE_TABLE_NAME record field of the IRD, which is a read-only field.</td>
</tr>
<tr>
<td>SQL_DESC_CASE_SENSITIVE (ODBC 1.0)</td>
<td>NumericAttributePtr</td>
<td>SQL_TRUE if the column is treated as case-sensitive for collations and comparisons. SQL_FALSE if the column is not treated as case-sensitive for collations and comparisons or is noncharacter.</td>
</tr>
<tr>
<td>SQL_DESC_CATALOG_NAME (ODBC 2.0)</td>
<td>CharacterAttributePtr</td>
<td>The catalog of the table that contains the column. The returned value is implementation-defined if the column is an expression or if the column is part of a view. If the data source does not support catalogs or the catalog name cannot be determined, an empty string is returned. This VARCHAR record field is not limited to 128 characters.</td>
</tr>
<tr>
<td>SQL_DESC_CONCISE_TYPE (ODBC 1.0)</td>
<td>NumericAttributePtr</td>
<td>The concise data type. For the datetime and interval data types, this field returns the concise data type; for example, SQL_TYPE_TIME or SQL_INTERVAL_YEAR. (For more information, see Data Type Identifiers and Descriptors in Appendix D: Data Types.) This information is returned from the SQL_DESC_CONCISE_TYPE record field of the IRD.</td>
</tr>
<tr>
<td>SQL_DESC_COUNT (ODBC 1.0)</td>
<td>NumericAttributePtr</td>
<td>The number of columns available in the result set. This returns 0 if there are no columns in the result set. The value in the ColumnNumber argument is ignored. This information is returned from the SQL_DESC_COUNT header field of the IRD.</td>
</tr>
<tr>
<td>SQL_DESC_DISPLAY_SIZE (ODBC 1.0)</td>
<td>NumericAttributePtr</td>
<td>Maximum number of characters required to display data from the column. For more information about display size, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.</td>
</tr>
</tbody>
</table>
| SQL_DESC_FIXED_PREC_SCALE  (ODBC 1.0) | NumericAttributePtr | SQL_TRUE if the column has a fixed precision and nonzero scale that are data source–specific. 
| | | SQL_FALSE if the column does not have a fixed precision and nonzero scale that are data source–specific. |
| SQL_DESC_LABEL  (ODBC 2.0) | CharacterAttributePtr | The column label or title. For example, a column named EmpName might be labeled Employee Name or might be labeled with an alias. If a column does not have a label, the column name is returned. If the column is unlabeled and unnamed, an empty string is returned. |
| SQL_DESC_LENGTH  (ODBC 3.0) | NumericAttributePtr | A numeric value that is either the maximum or actual character length of a character string or binary data type. It is the maximum character length for a fixed-length data type, or the actual character length for a variable-length data type. Its value always excludes the null-termination byte that ends the character string. This information is returned from the SQL_DESC_LENGTH record field of the IRD. For more information about length, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types. |
| SQL_DESC_LITERAL_PREFIX  (ODBC 3.0) | CharacterAttributePtr | This VARCHAR(128) record field contains the character or characters that the driver recognizes as a prefix for a literal of this data type. This field contains an empty string for a data type for which a literal prefix is not applicable. For more information, see Literal Prefixes and Suffixes. |
| SQL_DESC_LITERAL_SUFFIX  (ODBC 3.0) | CharacterAttributePtr | This VARCHAR(128) record field contains the character or characters that the driver recognizes as a suffix for a literal of this data type. This field contains an empty string for a data type for which a literal suffix is not applicable. For more information, see Literal Prefixes and Suffixes. |
| SQL_DESC_LOCAL_TYPE_NAME (ODBC 3.0) | CharacterAttributePtr | This VARCHAR(128) record field contains any localized (native language) name for the data type that may be different from the regular name of the data type. If there is no localized name, then an empty string is returned. This field is for display purposes only. The character set of the string is locale-dependent and is typically the default character set of the server. |
| SQL_DESC_NAME (ODBC 3.0) | CharacterAttributePtr | The column alias, if it applies. If the column alias does not apply, the column name is returned. In either case, SQL_DESC_UNNAMED is set to SQL_NAMED. If there is no column name or a column alias, an empty string is returned and SQL_DESC_UNNAMED is set to SQL_UNNAMED. This information is returned from the SQL_DESC_NAME record field of the IRD. |
| SQL_DESC_NULLABLE (ODBC 3.0) | NumericAttributePtr | SQL_NULLABLE if the column can have NULL values; SQL_NO_NULLS if the column does not have NULL values; or SQL_NULLABLE_UNKNOWN if it is not known whether the column accepts NULL values. This information is returned from the SQL_DESC_NULLABLE record field of the IRD. |
| SQL_DESC_NUM_PREC_RADIX (ODBC 3.0) | NumericAttributePtr | If the data type in the SQL_DESC_TYPE field is an approximate numeric data type, this SQLINTEGER field contains a value of 2 because the SQL_DESC_PRECISION field contains the number of bits. If the data type in the SQL_DESC_TYPE field is an exact numeric data type, this field contains a value of 10 because the SQL_DESC_PRECISION field contains the number of decimal digits. This field is set to 0 for all non-numeric data types. |
| SQL_DESC_OCTET_LENGTH (ODBC 3.0) | NumericAttributePtr | The length, in bytes, of a character string or binary data type. For fixed-length character or binary types, this is the actual length in bytes. For variable-length character or binary |
types, this is the maximum length in bytes. This value does not include the null terminator.

This information is returned from the SQL_DESC_OCTET_LENGTH record field of the IRD.

For more information about length, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

<table>
<thead>
<tr>
<th>SQL_DESC_PRECISION (ODBC 3.0)</th>
<th>NumericAttributePtr</th>
</tr>
</thead>
</table>
| A numeric value that for a numeric data type denotes the applicable precision. For data types SQL_TYPE_TIME, SQL_TYPE_TIMESTAMP, and all the interval data types that represent a time interval, its value is the applicable precision of the fractional seconds component. This information is returned from the SQL_DESC_PRECISION record field of the IRD.

<table>
<thead>
<tr>
<th>SQL_DESC_SCALE (ODBC 3.0)</th>
<th>NumericAttributePtr</th>
</tr>
</thead>
</table>
| A numeric value that is the applicable scale for a numeric data type. For DECIMAL and NUMERIC data types, this is the defined scale. It is undefined for all other data types. This information is returned from the SCALE record field of the IRD.

<table>
<thead>
<tr>
<th>SQL_DESC_SCHEMA_NAME (ODBC 2.0)</th>
<th>CharacterAttributePtr</th>
</tr>
</thead>
</table>
| The schema of the table that contains the column. The returned value is implementation-defined if the column is an expression or if the column is part of a view. If the data source does not support schemas or the schema name cannot be determined, an empty string is returned. This VARCHAR record field is not limited to 128 characters.

<table>
<thead>
<tr>
<th>SQL_DESC_SEARCHABLE (ODBC 1.0)</th>
<th>NumericAttributePtr</th>
</tr>
</thead>
</table>
| SQL_PRED_NONE if the column cannot be used in a WHERE clause. (This is the same as the SQL_UNSEARCHABLE value in ODBC 2.x.) SQL_PRED_CHAR if the column can be used in a WHERE clause but only with the LIKE predicate. (This is the same as the SQL_LIKE_ONLY value in ODBC...
2. SQL_PRED_BASIC if the column can be used in a WHERE clause with all the comparison operators except LIKE. (This is the same as the SQL_EXCEPT_LIKE value in ODBC 2.x.)

SQL_PRED_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.

Columns of type SQL_LONGVARCHAR and SQL_LONGVARBINARY usually return SQL_PRED_CHAR.

<table>
<thead>
<tr>
<th><strong>SQL_DESC_TABLE_NAME (ODBC 2.0)</strong></th>
<th><strong>CharacterAttributePtr</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The name of the table that contains the column. The returned value is implementation-defined if the column is an expression or if the column is part of a view.</td>
<td></td>
</tr>
<tr>
<td>If the table name cannot be determined, an empty string is returned.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>SQL_DESC_TYPE (ODBC 3.0)</strong></th>
<th><strong>NumericAttributePtr</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A numeric value that specifies the SQL data type.</td>
<td></td>
</tr>
<tr>
<td>When ColumnNumber is equal to 0, SQL_BINARY is returned for variable-length bookmarks and SQL_INTEGER is returned for fixed-length bookmarks.</td>
<td></td>
</tr>
<tr>
<td>For the datetime and interval data types, this field returns the verbose data type: SQL_DATETIME or SQL_INTERVAL. (For more information, see Data Type Identifiers and Descriptors in Appendix D: Data Types.</td>
<td></td>
</tr>
<tr>
<td>This information is returned from the SQL_DESC_TYPE record field of the IRD.</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

To work against ODBC 2.x drivers, use SQL_DESC_CONCISE_TYPE instead.
If the type is unknown, an empty string is returned.

**SQL_DESC_UNNAMED (ODBC 3.0)**

| NumericAttributePtr | SQL_NAMED or SQL_UNNAMED. If the SQL_DESC_NAME field of the IRD contains a column alias or a column name, SQL_NAMED is returned. If there is no column name or column alias, SQL_UNNAMED is returned. This information is returned from the SQL_DESC_UNNAMED record field of the IRD. |

**SQL_DESC_UNSIGNED (ODBC 1.0)**

| NumericAttributePtr | SQL_TRUE if the column is unsigned (or not numeric). SQL_FALSE if the column is signed. |

**SQL_DESC_UPDATABLE (ODBC 1.0)**

| NumericAttributePtr | Column is described by the values for the defined constants:

- SQL_ATTR_READONLY
- SQL_ATTR_WRITE
- SQL_ATTR_READWRITE_UNKNOWN

SQL_DESC_UPDATABLE describes the updatability of the column in the result set, not the column in the base table. The updatability of the base column on which the result set column is based may be different from the value in this field. Whether a column is updatable can be based on the data type, user privileges, and the definition of the result set itself. If it is unclear whether a column is updatable, SQL_ATTR_READWRITE_UNKNOWN should be returned. |

**SQLColAttribute** is an extensible alternative to **SQLDescribeCol**. **SQLDescribeCol** returns a fixed set of descriptor information based on ANSI-89 SQL. **SQLColAttribute** allows access to the more extensive set of descriptor information available in ANSI SQL-92 and DBMS vendor extensions.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic</td>
<td>Function</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td><strong>SQLBindCol Function</strong></td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td><strong>SQLCancel Function</strong></td>
</tr>
<tr>
<td>Returning information about a column in a result set</td>
<td><strong>SQLDescribeCol Function</strong></td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td><strong>SQLFetchScroll Function</strong></td>
</tr>
<tr>
<td>Fetching multiple rows of data</td>
<td><strong>SQLFetch Function</strong></td>
</tr>
</tbody>
</table>

The following sample code does not free handles and connections. See **SQLFreeHandle Function**, Sample ODBC Program, and **SQLFreeStmt Function** for code samples to free handles and statements.

```c
// SQLColAttribute.cpp
// compile with: user32.lib odbc32.lib

#define UNICODE

#include <windows.h>
#include <sqlext.h>
#include <strsafe.h>

struct DataBinding {
    SQLSMALLINT TargetType;
    SQLPOINTER TargetValuePtr;
    SQLINTEGER BufferLength;
    SQLLEN StrLen_or_Ind;
};

void printStatementResult(SQLHSTMT hstmt) {
    int bufferSize = 1024, i;
    SQLRETURN retCode;
    SQLSMALLINT numColumn = 0, bufferLenUsed;
    SQLPOINTER* columnLabels = (SQLPOINTER *)malloc( numColumn * sizeof(SQLPOINTER*) );
    struct DataBinding* columnData = (struct DataBinding*)malloc( numColumn * sizeof(struct

retCode = SQLNumResultCols(hstmt, &numColumn);

printf( "Columns from that table:\n" );
for ( i = 0 ; i < numColumn ; i++ ) {
    columnLabels[i] = (SQLPOINTER)malloc( bufferSize*sizeof(char) );

    retCode = SQLColAttribute(hstmt, (SQLUSMALLINT)i + 1, SQL_DESC_LABEL, columnLabels[i]
    wprintf( L"Column %d: %s\n", i, (wchar_t*)columnLabels[i] );
}

// allocate memory for the binding
for ( i = 0 ; i < numColumn ; i++ ) {
    columnData[i].TargetType = SQL_C_CHAR;
    columnData[i].BufferLength = (bufferSize+1);
    columnData[i].TargetValuePtr = malloc( sizeof(unsigned char)*columnData[i].BufferLen
```
// setup the binding
for ( i = 0 ; i < numColumn ; i++ ) {
    retCode = SQLBindCol(hstmt, (SQLUSMALLINT)i + 1, columnData[i].TargetType,
                            columnData[i].TargetValuePtr, columnData[i].BufferLength, &(columnData[i].StrLen_)
}

printf( "Data from that table:\n" );
// fetch the data and print out the data
for ( retCode = SQLFetch(hstmt); retCode == SQL_SUCCESS || retCode == SQL_SUCCESS_WITH
int j;
for ( j = 0 ; j < numColumn ; j++ )
    wprintf("%s: %hs\n", columnLabels[j], columnData[j].TargetValuePtr );
printf("\n");
}
printf("\n");
}

int main()
{
    int bufferSize = 1024, i, count = 1, numCols = 5;
    wchar_t firstTableName[1024], *dbname = (wchar_t*)malloc( sizeof(wchar_t)*bufferSize
HWND desktopHandle = GetDesktopWindow();  // desktop's window handle
SQLWCHAR connStrBuffer[1024];
SQLMALLINT connStrBufferLen, bufferLen;
SQLRETURN retCode;
SQLHENV henv = NULL;  // Environment
SQLHDBC hdc = NULL;  // Connection handle
SQLHSTMT hstmt = NULL;  // Statement handle

struct DataBinding* catalogResult = (struct DataBinding*) malloc( numCols * sizeof(struct
SQLWCHAR* selectAllQuery = (SQLWCHAR*)malloc( sizeof(SQLWCHAR) * bufferSize );

// connect to database
retCode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
retCode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLCHAR*)(void*)SQL_OV_ODBC3, -1
retCode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdb);
retCode = SQLGetInfo(hdb, SQL_DATABASE_NAME, db_name,  SQL_DBNAME_BUFFER_LEN, &bufferLen);
retCode = SQLDriverConnect(hdb, desktopHandle, L"Driver={SQL Server}", SQL_NTS, connStr
retCode = SQLAllocHandle(SQL_HANDLE_STMT, hdb, &hstmt);

// display the database information
retCode = SQLGetInfo(hdbc, SQL_DATABASE_NAME, db_name, (SQLSMALLINT)bufferSize, (SQLSMAL
retCode = SQLGetInfo(hdbc, SQL_USER_NAME, user_name, (SQLSMALLINT)bufferSize, &bufferLen
for ( i = 0 ; i < numCols ; i++ ) {
    catalogResult[i].TargetType = SQL_C_CHAR;
    catalogResult[i].BufferLength = (bufferSize + 1);
    catalogResult[i].TargetValuePtr = malloc( sizeof(unsigned char)*catalogResult[i].Buf
}

// Set up the binding. This can be used even if the statement is closed by closeStatementHandle
for ( i = 0 ; i < numCols ; i++ )
    retCode = SQLBindCol(hstmt, (SQLUSMALLINT)i + 1, catalogResult[i].TargetType, catalo
retCode = SQLGetInfo(hstmt, SQL_ALL_CATALOGS, SQL_NTS, L"", SQL_NTS, L"", SQL_ALL_CATALOGS
retCode = SQLTables( hstmt, (SQLCHAR*)SQL_ALL_CATALOGS, SQL_NTS, L"", SQL_NTS, L"", SQL_TABLES
retCode = SQLGetInfo(hstmt, SQL_ERROR, &error);
if ( retCode == SQL_NO_DATA_FOUND )
    printf("Table is empty!\n");
else
    printf("Table not found\n");
}

// close resources
SQLFreeStmt(hstmt, SQL_CLOSE);
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
SQLFreeHandle(SQL_HANDLE_DBC, hdb);
SQLFreeHandle(SQL_HANDLE_ENV, henv);
}

int main()
{
    int bufferSize = 1024, i, count = 1, numCols = 5;
    wchar_t firstTableName[1024], *dbname = (wchar_t*)malloc( sizeof(wchar_t)*bufferSize
HWND desktopHandle = GetDesktopWindow();  // desktop's window handle
SQLWCHAR connStrBuffer[1024];
SQLMALLINT connStrBufferLen, bufferLen;
SQLRETURN retCode;
SQLHENV henv = NULL;  // Environment
SQLHDBC hdc = NULL;  // Connection handle
SQLHSTMT hstmt = NULL;  // Statement handle

struct DataBinding* catalogResult = (struct DataBinding*) malloc( numCols * sizeof(struct
SQLWCHAR* selectAllQuery = (SQLWCHAR*)malloc( sizeof(SQLWCHAR) * bufferSize );

// connect to database
retCode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
retCode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLCHAR*)(void*)SQL_OV_ODBC3, -1
retCode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdb);
retCode = SQLGetInfo(hdb, SQL_DATABASE_NAME, db_name,  SQL_DBNAME_BUFFER_LEN, &bufferLen);
retCode = SQLDriverConnect(hdb, desktopHandle, L"Driver={SQL Server}", SQL_NTS, connStr
retCode = SQLAllocHandle(SQL_HANDLE_STMT, hdb, &hstmt);

// display the database information
retCode = SQLGetInfo(hdbc, SQL_DATABASE_NAME, db_name, (SQLSMALLINT)bufferSize, (SQLSMAL
retCode = SQLGetInfo(hdbc, SQL_USER_NAME, user_name, (SQLSMALLINT)bufferSize, &bufferLen
for ( i = 0 ; i < numCols ; i++ ) {
    catalogResult[i].TargetType = SQL_C_CHAR;
    catalogResult[i].BufferLength = (bufferSize + 1);
    catalogResult[i].TargetValuePtr = malloc( sizeof(unsigned char)*catalogResult[i].Buf
}

// Set up the binding. This can be used even if the statement is closed by closeStatementHandle
for ( i = 0 ; i < numCols ; i++ )
    retCode = SQLBindCol(hstmt, (SQLUSMALLINT)i + 1, catalogResult[i].TargetType, catalo
retCode = SQLGetInfo(hstmt, SQL_ALL_CATALOGS, SQL_NTS, L"", SQL_NTS, L"", SQL_ALL_CATALOGS
retCode = SQLTables( hstmt, (SQLCHAR*)SQL_ALL_CATALOGS, SQL_NTS, L"", SQL_NTS, L"", SQL_TABLES
retCode = SQLGetInfo(hstmt, SQL_ERROR, &error);
if ( retCode == SQL_NO_DATA_FOUND )
    printf("Table is empty!\n");
else
    printf("Table not found\n");
}

// close resources
SQLFreeStmt(hstmt, SQL_CLOSE);
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
SQLFreeHandle(SQL_HANDLE_DBC, hdb);
SQLFreeHandle(SQL_HANDLE_ENV, henv);
}
SQLColAttributes Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

Summary
In ODBC 3.x, the ODBC 2.0 function `SQLColAttributes` has been replaced by `SQLColAttribute`. For more information, see `SQLColAttribute Function`.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
</table>
For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.

See ODBC 64-Bit Information, if your application will run on a 64-bit operating system.

See Also
ODBC API Reference
ODBC Header Files
**Conformance**

Version Introduced: ODBC 1.0 Standards Compliance: ODBC

**Summary**

`SQLColumnPrivileges` returns a list of columns and associated privileges for the specified table. The driver returns the information as a result set on the specified `StatementHandle`.

**Syntax**

```c
SQLRETURN SQLColumnPrivileges(
    SQLHSTMT    StatementHandle,
    SQLCHAR    *    CatalogName,
    SQLSMALLINT NameLength1,
    SQLCHAR    *    SchemaName,
    SQLSMALLINT NameLength2,
    SQLCHAR    *    TableName,
    SQLSMALLINT NameLength3,
    SQLCHAR    *    ColumnName,
    SQLSMALLINT NameLength4);
```

**Arguments**

*StatementHandle*

[Input] Statement handle.

*CatalogName*

[Input] Catalog name. If a driver supports names for some catalogs but not for others, such as when the driver retrieves data from different DBMSs, an empty string (""") denotes those catalogs that do not have names. `CatalogName` cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, `CatalogName` is treated as an identifier and its case is not significant. If it is SQL_FALSE, `CatalogName` is an ordinary argument; it is treated literally, and its case is significant. For more information, see Arguments in Catalog Functions.

*NameLength1*

[Input] Length in characters of `*CatalogName`.

*SchemaName*

[Input] Schema name. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string (""") denotes those tables that do not have schemas. `SchemaName` cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, `SchemaName` is treated as an identifier. If it is SQL_FALSE, `SchemaName` is an ordinary argument; it is treated literally, and its case is significant.

*NameLength2*

[Input] Length in characters of `*SchemaName`.
**TableName**

[Input] Table name. This argument cannot be a null pointer. *TableName* cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *TableName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *TableName* is an ordinary argument; it is treated literally, and its case is significant.

**NameLength3**

[Input] Length in characters of *TableName*.

**ColumnName**

[Input] String search pattern for column names.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *ColumnName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *ColumnName* is a pattern value argument; it is treated literally, and its case is significant.

**NameLength4**

[Input] Length in characters of *ColumnName*.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When *SQLColumnPrivileges* returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling *SQLGetDiagRec* with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by *SQLColumnPrivileges* and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>A cursor was open on the StatementHandle, and <em>SQLFetch</em> or <em>SQLFetchScroll</em> had been called. This error is returned by the Driver Manager if <em>SQLFetch</em> or <em>SQLFetchScroll</em> has not returned SQL_NO_DATA, and is returned by the driver if <em>SQLFetch</em> or <em>SQLFetchScroll</em> has returned SQL_NO_DATA.</td>
</tr>
</tbody>
</table>
A cursor was open on the **StatementHandle**, but **SQLFetch** or **SQLFetchScroll** had not been called.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40001</td>
<td>Serialization failure The transaction was rolled back due to a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <strong>SQLGetDiagRec</strong> in the <em>MessageText</em> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled Asynchronous processing was enabled for the <strong>StatementHandle</strong>. The function was called, and before it completed execution, <strong>SQLCancel</strong> or <strong>SQLCancelHandle</strong> was called on the <strong>StatementHandle</strong>. Then the function was called again on the <strong>StatementHandle</strong>. The function was called, and before it completed execution, <strong>SQLCancel</strong> or <strong>SQLCancelHandle</strong> was called on the <strong>StatementHandle</strong> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer The <strong>TableName</strong> argument was a null pointer. The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the <strong>CatalogName</strong> argument was a null pointer, and the SQL_CATALOG_NAME InfoType returns that catalog names are supported. (DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the <strong>SchemaName</strong> or <strong>ColumnName</strong> argument was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error (DM) An asynchronously executing function was called for the connection handle that is associated with the <strong>StatementHandle</strong>. This asynchronous function was still executing when this function was called. (DM) <strong>SQLExecute</strong>, <strong>SQLExecDirect</strong>, or <strong>SQLMoreResults</strong> was called for the <strong>StatementHandle</strong> and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. (DM) An asynchronously executing function (not this one) was called for the <strong>StatementHandle</strong> and was still executing when this function was called. (DM) <strong>SQLExecute</strong>, <strong>SQLExecDirect</strong>, <strong>SQLBulkOperations</strong>, or <strong>SQLSetPos</strong> was called for the <strong>StatementHandle</strong> and...</td>
</tr>
<tr>
<td>Code</td>
<td>Category</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
</tr>
</tbody>
</table>
| HYC00 | Optional feature not implemented | A catalog name was specified, and the driver or data source does not support catalogs.  
A schema name was specified, and the driver or data source does not support schemas.  
A string search pattern was specified for the column name, and the data source does not support search patterns for that argument.  
The combination of the current settings of the SQL_CONCURRNCY and SQL_CURSOR_TYPE statement attributes was not supported by the driver or data source.  
The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks. |
| HYT00 | Timeout expired | The query timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT. |
| HYT01 | Connection timeout expired | The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT. |
| IM001 | Driver does not support this function | (DM) The driver associated with the StatementHandle does not support the function. |
Polling is disabled in asynchronous notification mode

Whenever the notification model is used, polling is disabled.

SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.

If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.

Comments

**SQLColumnPrivileges** returns the results as a standard result set, ordered by TABLE_CAT, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, and PRIVILEGE.

**Note**

**SQLColumnPrivileges** might not return privileges for all columns. For example, a driver might not return information about privileges for pseudo-columns, such as Oracle ROWID. Applications can use any valid column, regardless of whether it is returned by SQLColumnPrivileges.

The lengths of VARCHAR columns are not shown in the table; the actual lengths depend on the data source. To determine the actual lengths of the CATALOG_NAME, SCHEMA_NAME, TABLE_NAME, and COLUMN_NAME columns, an application can call SQLGetInfo with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

**Note**

For more information about the general use, arguments, and returned data of ODBC catalog functions, see Catalog Functions.

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>TABLE_CAT</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>TABLE_SCHEMA</td>
</tr>
</tbody>
</table>

The following table lists the columns in the result set. Additional columns beyond column 8 (IS_GRANTABLE) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set rather than specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.
<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_CAT (ODBC 1.0)</td>
<td>1</td>
<td>Varchar</td>
<td>Catalog identifier; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those tables that do not have catalogs.</td>
</tr>
<tr>
<td>TABLE_SCHEM (ODBC 1.0)</td>
<td>2</td>
<td>Varchar</td>
<td>Schema identifier; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those tables that do not have schemas.</td>
</tr>
<tr>
<td>TABLE_NAME (ODBC 1.0)</td>
<td>3</td>
<td>Varchar</td>
<td>Table identifier.</td>
</tr>
<tr>
<td>COLUMN_NAME (ODBC 1.0)</td>
<td>4</td>
<td>Varchar</td>
<td>Column name. The driver returns an empty string for a column that does not have a name.</td>
</tr>
<tr>
<td>GRANTOR (ODBC 1.0)</td>
<td>5</td>
<td>Varchar</td>
<td>Name of the user who granted the privilege; NULL if not applicable to the data source. For all rows in which the value in the GRANTEE column is the owner of the object, the GRANTOR column will be &quot;_SYSTEM&quot;.</td>
</tr>
<tr>
<td>GRANTEE (ODBC 1.0)</td>
<td>6</td>
<td>Varchar</td>
<td>Name of the user to whom the privilege was granted.</td>
</tr>
<tr>
<td>PRIVILEGE (ODBC 1.0)</td>
<td>7</td>
<td>Varchar</td>
<td>Identifies the column privilege. May be one of the following (or others supported by the data source when implementation-defined):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SELECT: The grantee is permitted to retrieve data for the column.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INSERT: The grantee is permitted to provide data for the column in new rows that are inserted into the associated table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>UPDATE: The grantee is permitted to update data in the column.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REFERENCES: The grantee is permitted to refer to the column within a constraint (for example, a unique, referential, or table check constraint).</td>
</tr>
<tr>
<td>IS_GRANTABLE (ODBC 1.0)</td>
<td>8</td>
<td>Varchar</td>
<td>Indicates whether the grantee is permitted to grant the privilege to other users; &quot;YES&quot;, &quot;NO&quot;, or &quot;NULL&quot; if unknown or not applicable to the data source.</td>
</tr>
</tbody>
</table>
A privilege is either grantable or not grantable, but not both. The result set returned by `SQLColumnPrivileges` will never contain two rows for which all columns except the `IS_GRANTABLE` column contain the same value.

### Code Example

For a code example of a similar function, see `SQLColumns Function`.

### Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td><code>SQLBindCol Function</code></td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td><code>SQLCancel Function</code></td>
</tr>
<tr>
<td>Returning the columns in a table or tables</td>
<td><code>SQLColumns Function</code></td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td><code>SQLFetchScroll Function</code></td>
</tr>
<tr>
<td>Fetching multiple rows of data</td>
<td><code>SQLFetch Function</code></td>
</tr>
<tr>
<td>Returning privileges for a table or tables</td>
<td><code>SQLTablePrivileges Function</code></td>
</tr>
<tr>
<td>Returning a list of tables in a data source</td>
<td><code>SQLTables Function</code></td>
</tr>
</tbody>
</table>

### See Also

- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft

## `SQLColumns` Function

### Conformance

Version Introduced: ODBC 1.0 Standards Compliance: Open Group

### Summary

`SQLColumns` returns the list of column names in specified tables. The driver returns this
information as a result set on the specified StatementHandle.

**Syntax**

```c
SQLRETURN SQLColumns(
    SQLHSTMT StatementHandle,
    SQLCHAR * CatalogName,
    SQLSMALLINT NameLength1,
    SQLCHAR * SchemaName,
    SQLSMALLINT NameLength2,
    SQLCHAR * TableName,
    SQLSMALLINT NameLength3,
    SQLCHAR * ColumnName,
    SQLSMALLINT NameLength4);
```

**Arguments**

**StatementHandle**

[Input] Statement handle.

**CatalogName**

[Input] Catalog name. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") indicates those tables that do not have catalogs. CatalogName cannot contain a string search pattern.

**Note**

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, CatalogName is treated as an identifier and its case is not significant. If it is SQL_FALSE, CatalogName is an ordinary argument; it is treated literally, and its case is significant. For more information, see Arguments in Catalog Functions.

**NameLength1**

[Input] Length in characters of *CatalogName.

**SchemaName**

[Input] String search pattern for schema names. If a driver supports schema for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") indicates those tables that do not have schemas.

**Note**

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, SchemaName is treated as an identifier and its case is not significant. If it is SQL_FALSE, SchemaName is a pattern value argument; it is treated literally, and its case is significant.

**NameLength2**
Input] Length in characters of *SchemaName.

**TableName**
[Input] String search pattern for table names.

**Note**
If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *TableName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *TableName* is a pattern value argument; it is treated literally, and its case is significant.

**NameLength3**
[Input] Length in characters of *TableName*.

**ColumnName**
[Input] String search pattern for column names.

**Note**
If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *ColumnName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *ColumnName* is a pattern value argument; it is treated literally, and its case is significant.

**NameLength4**
[Input] Length in characters of *ColumnName*.

**Returns**
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When **SQLColumns** returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling **SQLGetDiagRec** with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values typically returned by **SQLColumns** and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link</td>
<td>The communication link between the driver and the data</td>
</tr>
<tr>
<td>Failure Code</td>
<td>Description</td>
<td>Error Message</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>failure</td>
<td>source to which the driver was connected failed before the function completed processing.</td>
<td></td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state A cursor was open on the <code>StatementHandle</code>, and <code>SQLFetch</code> or <code>SQLFetchScroll</code> had been called. This error is returned by the Driver Manager if <code>SQLFetch</code> or <code>SQLFetchScroll</code> has not returned <code>SQL_NO_DATA</code>, and is returned by the driver if <code>SQLFetch</code> or <code>SQLFetchScroll</code> has returned <code>SQL_NO_DATA</code>. A cursor was open on the <code>StatementHandle</code> but <code>SQLFetch</code> or <code>SQLFetchScroll</code> had not been called.</td>
<td></td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure The transaction was rolled back because of a resource deadlock with another transaction.</td>
<td></td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>General error An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
<td></td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
<td></td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled Asynchronous processing was enabled for the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code>. Then the function was called again on the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code> from a different thread in a multithread application.</td>
<td></td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the CatalogName argument was a null pointer, and the SQL_CATALOG_NAME InfoType returns that catalog names are supported. (DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the SchemaName, TableName, or ColumnName argument was a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error (DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when the SQLColumns function was called. (DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the <code>StatementHandle</code> and returned SQL_PARAM_DATA_AVAILABLE. This function was called</td>
<td></td>
</tr>
</tbody>
</table>
before data was retrieved for all streamed parameters.  
(DM) An asynchronously executing function (not this one)  
was called for the `StatementHandle` and was still executing  
when this function was called.  
(DM) `SQLExecute`, `SQLExecDirect`, `SQLBulkOperations`,  
or `SQLSetPos` was called for the `StatementHandle` and  
returned `SQL_NEED_DATA`. This function was called before  
data was sent for all data-at-execution parameters or  
columns.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td></td>
<td>(DM) The value of one of the name length arguments was less than 0 but not equal to SQL_NTS.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td></td>
<td>(DM) For more information about suspended state, see <code>SQLEndTran Function</code>.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Issue</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
<td></td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
<td></td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
<td></td>
</tr>
<tr>
<td>IM018</td>
<td>SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.</td>
<td></td>
</tr>
</tbody>
</table>

**Comments**

This function typically is used before statement execution to retrieve information about columns for a table or tables from the data source's catalog. **SQLColumns** can be used to retrieve data for all types of items returned by **SQLTables**. In addition to base tables, this may include (but is not limited to) views, synonyms, system tables, and so on. By contrast, the functions **SQLColAttribute** and **SQLDescribeCol** describe the columns in a result set and the function **SQLNumResultCols** returns the number of columns in a result set. For more information, see [Uses of Catalog Data](#).

**Note**

For more information about the general use, arguments, and returned data of ODBC catalog functions, see [Catalog Functions](#).

**SQLColumns** returns the results as a standard result set, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and ORDINAL_POSITION.

**Note**

When an application works with an ODBC 2.x driver, no ORDINAL_POSITION column is returned in the result set. As a result, when working with ODBC 2.x drivers, the order of the columns in the column list returned by **SQLColumns** is not necessarily the same as the order of the columns returned when the application performs a SELECT statement on all columns in that table.

**Note**

**SQLColumns** might not return all columns. For example, a driver might not return information about pseudo-columns, such as Oracle ROWID. Applications can use any valid column, whether it is returned by **SQLColumns**.
Some columns that can be returned by SQLStatistics are not returned by SQLColumns. For example, SQLColumns does not return the columns in an index created over an expression or filter, such as SALARY + BENEFITS or DEPT = 0012.

The lengths of VARCHAR columns are not shown in the table; the actual lengths depend on the data source. To determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns, an application can call SQLGetInfo with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>TABLE_CAT</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>TABLE_SCHEM</td>
</tr>
<tr>
<td>PRECISION</td>
<td>COLUMN_SIZE</td>
</tr>
<tr>
<td>LENGTH</td>
<td>BUFFER_LENGTH</td>
</tr>
<tr>
<td>SCALE</td>
<td>DECIMAL_DIGITS</td>
</tr>
<tr>
<td>RADIX</td>
<td>NUM_PREC_RADIX</td>
</tr>
</tbody>
</table>

The following columns have been added to the result set returned by SQLColumns for ODBC 3.x:

| CHAR_OCTET_LENGTH       | ORDINAL_POSITION       |
|                        |                        |
| COLUMN_DEF              | SQL_DATA_TYPE          |
| IS_NULLABLE            | SQL_DATETIME_SUB       |

The following table lists the columns in the result set. Additional columns beyond column 18 (IS_NULLABLE) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set instead of specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_CAT (ODBC 1.0)</td>
<td>1</td>
<td>Varchar</td>
<td>Catalog name; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those tables that do not have</td>
</tr>
<tr>
<td>TABLE</td>
<td>DATA TYPE</td>
<td>SIZE</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td><strong>TABLE_SCHEM (ODBC 1.0)</strong></td>
<td>Varchar</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>TABLE_NAME (ODBC 1.0)</strong></td>
<td>Varchar</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td><strong>COLUMN_NAME (ODBC 1.0)</strong></td>
<td>Varchar</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>DATA_TYPE (ODBC 1.0)</strong></td>
<td>Smallint</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td><strong>TYPE_NAME (ODBC 1.0)</strong></td>
<td>Varchar</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td><strong>COLUMN_SIZE (ODBC 1.0)</strong></td>
<td>Integer</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td><strong>BUFFER_LENGTH (ODBC 1.0)</strong></td>
<td>Integer</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE SCHEM (ODBC 1.0)**

Schema name; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (""") for those tables that do not have schemas.

**TABLE_NAME (ODBC 1.0)**

Table name.

**COLUMN_NAME (ODBC 1.0)**

Column name. The driver returns an empty string for a column that does not have a name.

**DATA_TYPE (ODBC 1.0)**

SQL data type. This can be an ODBC SQL data type or a driver-specific SQL data type. For datetime and interval data types, this column returns the concise data type (such as SQL_TYPE_DATE or SQL_INTERVAL_YEAR_TO_MONTH, instead of the nonconcise data type such as SQL_DATETIME or SQL_INTERVAL). For a list of valid ODBC SQL data types, see SQL Data Types in Appendix D: Data Types. For information about driver-specific SQL data types, see the driver’s documentation.

The data types returned for ODBC 3.x and ODBC 2.x applications may be different. For more information, see Backward Compatibility and Standards Compliance.

**TYPE_NAME (ODBC 1.0)**

Data source–dependent data type name; for example, "CHAR", "VARCHAR", "MONEY", "LONG VARBINAR", or "CHAR ( ) FOR BIT DATA".

**COLUMN_SIZE (ODBC 1.0)**

If DATA_TYPE is SQL_CHAR or SQL_VARCHAR, this column contains the maximum length in characters of the column. For datetime data types, this is the total number of characters required to display the value when it is converted to characters. For numeric data types, this is either the total number of digits or the total number of bits allowed in the column, according to the NUM_PREC_RADIX column. For interval data types, this is the number of characters in the character representation of the interval literal (as defined by the interval leading precision, see Interval Data Type Length in Appendix D: Data Types). For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

**BUFFER_LENGTH (ODBC 1.0)**

The length in bytes of data transferred on an SQLGetData, SQLFetch, or SQLFetchScroll operation.
If SQL_C_DEFAULT is specified. For numeric data, this size may differ from the size of the data stored on the data source. This value might differ from COLUMN_SIZE column for character data. For more information about length, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

<table>
<thead>
<tr>
<th>DECIMAL_DIGITS (ODBC 1.0)</th>
<th>9</th>
<th>Smallint</th>
<th>The total number of significant digits to the right of the decimal point. For SQL_TYPE_TIME and SQL_TYPE_TIMESTAMP, this column contains the number of digits in the fractional seconds component. For the other data types, this is the decimal digits of the column on the data source. For interval data types that contain a time component, this column contains the number of digits to the right of the decimal point (fractional seconds). For interval data types that do not contain a time component, this column is 0. For more information about decimal digits, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types. NULL is returned for data types where DECIMAL_DIGITS is not applicable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_PREC_RADIX (ODBC 1.0)</td>
<td>10</td>
<td>Smallint</td>
<td>For numeric data types, either 10 or 2. If it is 10, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of decimal digits allowed for the column. For example, a DECIMAL(12,5) column would return a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 12, and a DECIMAL_DIGITS of 5; a FLOAT column could return a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 15, and a DECIMAL_DIGITS of NULL. If it is 2, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of bits allowed in the column. For example, a FLOAT column could return a RADIX of 2, a COLUMN_SIZE of 53, and a DECIMAL_DIGITS of NULL. NULL is returned for data types where NUM_PREC_RADIX is not applicable.</td>
</tr>
<tr>
<td>NULLABLE (ODBC 1.0)</td>
<td>11</td>
<td>Smallint</td>
<td>SQL_NO_NULLS if the column could not include NULL values. SQL_NULLABLE if the column accepts NULL values. SQL_NULLABLE_UNKNOWN if it is not known whether the column accepts NULL values. The value returned for this column differs from the value returned for the IS_NULLABLE column. The NULLABLE column indicates with certainty that a column can accept NULLs, but cannot indicate with certainty that a column does not accept NULLs. The</td>
</tr>
<tr>
<td>Column Name</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>IS_NULLABLE</td>
<td></td>
<td>The IS_NULLABLE column indicates with certainty that a column cannot accept NULLs, but cannot indicate with certainty that a column accepts NULLs.</td>
<td></td>
</tr>
<tr>
<td>REMARKS (ODBC 1.0)</td>
<td>12</td>
<td>Varchar                                                                 A description of the column.</td>
<td></td>
</tr>
<tr>
<td>COLUMN_DEF (ODBC 3.0)</td>
<td>13</td>
<td>Varchar                                                                 The default value of the column. The value in this column should be interpreted as a string if it is enclosed in quotation marks.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If NULL was specified as the default value, this column is the word NULL, not enclosed in quotation marks. If the default value cannot be represented without truncation, this column contains TRUNCATED, without enclosing single quotation marks. If no default value was specified, this column is NULL.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value of COLUMN_DEF can be used in generating a new column definition, except when it contains the value TRUNCATED.</td>
<td></td>
</tr>
<tr>
<td>SQL_DATA_TYPE (ODBC 3.0)</td>
<td>14</td>
<td>Smallint, not NULL                                                                 SQL data type, as it appears in the SQL_DESC_TYPE record field in the IRD. This can be an ODBC SQL data type or a driver-specific SQL data type. This column is the same as the DATA_TYPE column, except for datetime and interval data types. This column returns the nonconcise data type (such as SQL_DATETIME or SQL_INTERVAL), instead of the concise data type (such as SQL_TYPE_DATE or SQL_INTERVAL_YEAR_TO_MONTH) for datetime and interval data types. If this column returns SQL_DATETIME or SQL_INTERVAL, the specific data type can be determined from the SQL_DATETIME_SUB column. For a list of valid ODBC SQL data types, see SQL Data Types in Appendix D: Data Types. For information about driver-specific SQL data types, see the driver's documentation. The data types returned for ODBC 3.x and ODBC 2.x applications may be different. For more information, see Backward Compatibility and Standards Compliance.</td>
<td></td>
</tr>
<tr>
<td>SQL_DATETIME_SUB (ODBC 3.0)</td>
<td>15</td>
<td>Smallint                                                                 The subtype code for datetime and interval data types. For other data types, this column returns a NULL. For more information about datetime and interval subcodes, see &quot;SQL_DESC_DATETIME_INTERVAL_CODE&quot; in SQLSetDescField.</td>
<td></td>
</tr>
<tr>
<td>CHAR_OCTET_LENGTH (ODBC 3.0)</td>
<td>16</td>
<td>Integer                                                                 The maximum length in bytes of a character or binary data type column. For all other data types, this column returns a NULL.</td>
<td></td>
</tr>
<tr>
<td>ORDINAL POSITION (ODBC 3.0)</td>
<td>17</td>
<td>Integer not NULL</td>
<td>The ordinal position of the column in the table. The first column in the table is number 1.</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----</td>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IS_NULLABLE (ODBC 3.0)</td>
<td>18</td>
<td>Varchar</td>
<td>&quot;NO&quot; if the column does not include NULLs. &quot;YES&quot; if the column could include NULLs. This column returns a zero-length string if nullability is unknown. ISO rules are followed to determine nullability. An ISO SQL-compliant DBMS cannot return an empty string. The value returned for this column differs from the value returned for the NULLABLE column. (See the description of the NULLABLE column.)</td>
</tr>
</tbody>
</table>

**Code Example**

In the following example, an application declares buffers for the result set returned by `SQLColumns`. It calls `SQLColumns` to return a result set that describes each column in the `EMPLOYEE` table. It then calls `SQLBindCol` to bind the columns in the result set to the buffers. Finally, the application fetches each row of data with `SQLFetch` and processes it.

```c
// SQLColumns_Function.cpp
// compile with: ODBC32.lib
#include <windows.h>
#include <sqlext.h>
#define STR_LEN 128 + 1
#define REM_LEN 254 + 1

// Declare buffers for result set data
SQLCHAR szSchema[STR_LEN];
SQLCHAR szCatalog[STR_LEN];
SQLCHAR szColumnName[STR_LEN];
SQLCHAR szTableName[STR_LEN];
SQLCHAR szTypeName[STR_LEN];
SQLCHAR szRemarks[REM_LEN];
SQLCHAR szColumnDefault[STR_LEN];
SQLCHAR szIsNullable[STR_LEN];

SQLINTEGER ColumnSize;
SQLINTEGER BufferLength;
SQLINTEGER CharOctetLength;
SQLINTEGER OrdinalPosition;

SQLSMALLINT DataType;
SQLSMALLINT DecimalDigits;
SQLSMALLINT NumPrecRadix;
SQLSMALLINT Nullable;
```
int main() {
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLHSTMT hstmt = 0;
    SQLRETURN retcode;

    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)SQL_OV_ODBC3, 0);
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
    retcode = SQLConnect(hdbc, (SQLCHAR*)"Northwind", SQL_NTS, (SQLCHAR*)NULL, 0, NULL, NULL);
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    retcode = SQLColumns(hstmt, NULL, 0, NULL, 0, (SQLCHAR*)"CUSTOMERS", SQL_NTS, NULL, 0, NULL);

    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
        // Bind columns in result set to buffers
        SQLBindCol(hstmt, 1, SQL_C_CHAR, &szCatalog, STR_LEN, &cbCatalog);
        SQLBindCol(hstmt, 2, SQL_C_CHAR, &szSchema, STR_LEN, &cbSchema);
        SQLBindCol(hstmt, 3, SQL_C_CHAR, &szTableName, STR_LEN, &cbTableName);
        SQLBindCol(hstmt, 4, SQL_C_CHAR, &szColumnName, STR_LEN, &cbColumnName);
        SQLBindCol(hstmt, 5, SQL_C_SSHORT, &DataType, 0, &cbDataType);
        SQLBindCol(hstmt, 6, SQL_C_CHAR, &szTypeName, STR_LEN, &cbTypeName);
        SQLBindCol(hstmt, 7, SQL_C_SLONG, &ColumnSize, 0, &cbColumnSize);
        SQLBindCol(hstmt, 8, SQL_C_SLONG, &BufferLength, 0, &cbBufferLength);
        SQLBindCol(hstmt, 9, SQL_C_SLONG, &DecimalDigits, 0, &cbDecimalDigits);
        SQLBindCol(hstmt, 10, SQL_C_SSHORT, &NumPrecRadix, 0, &cbNumPrecRadix);
        SQLBindCol(hstmt, 11, SQL_C_CHAR, &szRemarks, REM_LEN, &cbRemarks);
        SQLBindCol(hstmt, 12, SQL_C_CHAR, &szColumnDefault, STR_LEN, &cbColumnDefault);
        SQLBindCol(hstmt, 13, SQL_C_CHAR, &szCharOctetLength, 0, &cbCharOctetLength);
        SQLBindCol(hstmt, 14, SQL_C_CHAR, &szOrdinalPosition, 0, &cbOrdinalPosition);
        SQLBindCol(hstmt, 15, SQL_C_CHAR, &szIsNullable, STR_LEN, &cbIsNullable);
    }
}
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Returning privileges for a column or columns</td>
<td>SQLColumnPrivileges Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Fetching multiple rows of data</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Returning columns that uniquely identify a row, or columns automatically updated by a transaction</td>
<td>SQLSpecialColumns Function</td>
</tr>
<tr>
<td>Returning table statistics and indexes</td>
<td>SQLStatistics Function</td>
</tr>
<tr>
<td>Returning a list of tables in a data source</td>
<td>SQLTables Function</td>
</tr>
<tr>
<td>Returning privileges for a table or tables</td>
<td>SQLTablePrivileges Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft
SQLCompleteAsync Function

Conformance
Version Introduced: ODBC 3.8
Standards Compliance: None

Summary
SQLCompleteAsync can be used to determine when an asynchronous function is complete using either notification- or polling-based processing. For more information about asynchronous operations, see Asynchronous Execution.

SQLCompleteAsync is only implemented in the ODBC Driver Manager.

In notification based asynchronous processing mode, SQLCompleteAsync must be called after the Driver Manager raises the event object used for notification. SQLCompleteAsync completes the asynchronous processing and the asynchronous function will generate a return code.

In polling based asynchronous processing mode, SQLCompleteAsync is an alternative to calling the original asynchronous function, without needing to specify the arguments in the original asynchronous function call. SQLCompleteAsync can be used regardless whether the ODBC Cursor Library is enabled.

Syntax

```vb
SQLRETURN SQLCompleteAsync(
    SQLSMALLINT HandleType,
    SQLHANDLE Handle,
    RETCODE * AsyncRetCodePtr);
```

Arguments

HandleType
[Input] The type of the handle on which to complete asynchronous processing. Valid values are SQL_HANDLE_DBC or SQL_HANDLE_STMT.

Handle
[Input] The handle on which to complete asynchronous processing. If Handle is not a valid handle of the type specified by HandleType, SQLCompleteAsync returns SQL_INVALID_HANDLE.

If Handle is not a valid handle of the type specified by HandleType, SQLCompleteAsync returns SQL_INVALID_HANDLE.

AsyncRetCodePtr
[Output] Pointer to a buffer that will contain the return code of the asynchronous API. If AsyncRetCodePtr is NULL, SQLCompleteAsync returns SQL_ERROR.
Returns

SQL_SUCCESS, SQL_ERROR, SQL_NO_DATA, or SQL_INVALID_HANDLE.

Diagnostics

If SQLCompleteAsync returns SQL_SUCCESS, an application should get the return code of the asynchronous function from the buffer pointed to by AsyncRetCodePtr. The associated SQLSTATE, if any, can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a statement handle or a HandleType of SQL_HANDLE_DBC and a connection handle. Those diagnostic records are associated with the asynchronous function, not this SQLCompleteAsync function.

SQLCompleteAsync returns a code other than SQL_SUCCESS to indicate that SQLCompleteAsync is not called correctly. SQLCompleteAsync will not post any diagnostic record in this case. Possible return codes are:

- SQL_INVALID_HANDLE: The handle indicated by HandleType and Handle is not a valid handle.
- SQL_ERROR: AsyncRetCodePtr is NULL or asynchronous processing is not enabled on the handle.
- SQL_NO_DATA: In notification mode, an asynchronous operation is not in progress or the Driver Manager has not notified the application. In polling mode, an asynchronous operation is not in progress.

Comments

In polling based asynchronous processing mode, AsyncRetCodePtr might be SQL_STILL_EXECUTING when SQLCompleteAsync returns SQL_SUCCESS. Application should keep polling until AsyncRetCodePtr is not SQL_STILL_EXECUTING. In notification based asynchronous processing mode, AsyncRetCodePtr will never be SQL_STILL_EXECUTING.

See Also

Asynchronous Execution (Polling Method)

© 2016 Microsoft

SQLConnect Function

Conformance
Summary

SQLConnect establishes connections to a driver and a data source. The connection handle references storage of all information about the connection to the data source, including status, transaction state, and error information.

Syntax

```c
SQLRETURN SQLConnect(
    SQLHDBC         ConnectionHandle,
    SQLCHAR         *  ServerName,
    SQLSMALLINT     NameLength1,
    SQLCHAR         *  UserName,
    SQLSMALLINT     NameLength2,
    SQLCHAR         *  Authentication,
    SQLSMALLINT     NameLength3);
```

Arguments

- **ConnectionHandle**
  [Input] Connection handle.

- **ServerName**
  [Input] Data source name. The data might be located on the same computer as the program, or on another computer somewhere on a network. For information about how an application chooses a data source, see Choosing a Data Source or Driver.

- **NameLength1**
  [Input] Length of *ServerName in characters.

- **UserName**
  [Input] User identifier.

- **NameLength2**
  [Input] Length of *UserName in characters.

- **Authentication**
  [Input] Authentication string (typically the password).

- **NameLength3**
  [Input] Length of *Authentication in characters.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_STILL_EXECUTING.
## Diagnostics

When `SQLConnect` returns `SQL_ERROR` or `SQL_SUCCESS_WITH_INFO`, an associated SQLSTATE value can be obtained by calling `SQLGetDiagRec` with a `HandleType` of `SQL_HANDLE_DBC` and a `Handle` of `ConnectionHandle`. The following table lists the SQLSTATE values typically returned by `SQLConnect` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is `SQL_ERROR`, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns <code>SQL_SUCCESS_WITH_INFO</code>.)</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed</td>
<td>The driver did not support the specified value of the <code>ValuePtr</code> argument in <code>SQLSetConnectAttr</code> and substituted a similar value. (Function returns <code>SQL_SUCCESS_WITH_INFO</code>.)</td>
</tr>
<tr>
<td>08001</td>
<td>Client unable to establish connection</td>
<td>The driver was unable to establish a connection with the data source.</td>
</tr>
<tr>
<td>08002</td>
<td>Connection name in use</td>
<td>(DM) The specified <code>ConnectionHandle</code> had already been used to establish a connection with a data source, and the connection was still open or the user was browsing for a connection.</td>
</tr>
<tr>
<td>08004</td>
<td>Server rejected the connection</td>
<td>The data source rejected the establishment of the connection for implementation-defined reasons.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was trying to connect failed before the function completed processing.</td>
</tr>
<tr>
<td>28000</td>
<td>Invalid authorization specification</td>
<td>The value specified for the argument <code>UserName</code> or the value specified for the argument <code>Authentication</code> violated restrictions defined by the data source.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <code>*MessageText</code> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>(DM) The Driver Manager was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <code>ConnectionHandle</code>. The <code>SQLConnect</code> function was called, and before it completed execution, <code>SQLCancelHandle</code> function was called on the <code>ConnectionHandle</code>, and then the <code>SQLConnect</code> function was called again on the <code>ConnectionHandle</code>. Or, the <code>SQLConnect</code> function was called, and before it completed execution, <code>SQLCancelHandle</code> was called on the <code>ConnectionHandle</code> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function (not this one) was called for the <code>ConnectionHandle</code> and was still executing when this function was called.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value specified for argument <code>NameLength1</code>, <code>NameLength2</code>, or <code>NameLength3</code> was less than 0 but not equal to SQL_NTS. (DM) The value specified for argument <code>NameLength1</code> exceeded the maximum length for a data source name.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the connection to the data source completed. The timeout period is set through <code>SQLSetConnectAttr</code>, <code>SQL_ATTR_LOGIN_TIMEOUT</code>.</td>
</tr>
<tr>
<td>HY114</td>
<td>Driver does not support connection level asynchronous function execution</td>
<td>(DM) The application enabled the asynchronous operation on the connection handle before making the connection. However, the driver does not support asynchronous operations on the connection handle.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>, <code>SQL_ATTR_CONNECTION_TIMEOUT</code>.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver specified by the data source name does not support the function.</td>
</tr>
<tr>
<td>IM002</td>
<td>Data source not found and no</td>
<td>(DM) The data source name specified in the</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IM003</td>
<td>Specified driver could not be connected to</td>
<td>(DM) The driver listed in the data source specification in system information was not found or could not be connected to for some other reason.</td>
</tr>
<tr>
<td>IM004</td>
<td>Driver's SQLAllocHandle on SQL_HANDLE_ENV failed</td>
<td>(DM) During <code>SQLConnect</code>, the Driver Manager called the driver's <code>SQLAllocHandle</code> function with a <code>HandleType</code> of SQL_HANDLE_ENV and the driver returned an error.</td>
</tr>
<tr>
<td>IM005</td>
<td>Driver's SQLAllocHandle on SQL_HANDLE_DBC failed</td>
<td>(DM) During <code>SQLConnect</code>, the Driver Manager called the driver's <code>SQLAllocHandle</code> function with a <code>HandleType</code> of SQL_HANDLE_DBC and the driver returned an error.</td>
</tr>
<tr>
<td>IM006</td>
<td>Driver's SQLSetConnectAttr failed</td>
<td>During <code>SQLConnect</code>, the Driver Manager called the driver's <code>SQLSetConnectAttr</code> function and the driver returned an error. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>IM009</td>
<td>Unable to connect to translation DLL</td>
<td>The driver was unable to connect to the translation DLL that was specified for the data source.</td>
</tr>
<tr>
<td>IM010</td>
<td>Data source name too long</td>
<td>(DM) <em>ServerName</em> was longer than SQL_MAX_DSN_LENGTH characters.</td>
</tr>
<tr>
<td>IM014</td>
<td>The specified DSN contains an architecture mismatch between the Driver and Application</td>
<td>(DM) 32-bit application uses a DSN connecting to a 64-bit driver; or vice versa.</td>
</tr>
<tr>
<td>IM015</td>
<td>Driver's SQLConnect on SQL_HANDLE_DBC_INFO_HANDLE failed</td>
<td>If a driver returns SQL_ERROR, the Driver Manager will return SQL_ERROR to the application and the connection will fail. For more information about SQL_HANDLE_DBC_INFO_TOKEN, see Developing Connection-Pool Awareness in an ODBC Driver.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td>SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>
When the driver does not support asynchronous notification, you cannot set SQL_ATTR_ASYNC_DBC_EVENT or SQL_ATTR_ASYNC_DBC RETCODE_PTR.

Comments

For information about why an application uses SQLConnect, see Connecting with SQLConnect.

The Driver Manager does not connect to a driver until the application calls a function (SQLConnect, SQLDriverConnect, or SQLBrowseConnect) to connect to the driver. Until that point, the Driver Manager works with its own handles and manages connection information. When the application calls a connection function, the Driver Manager checks whether a driver is currently connected to for the specified ConnectionHandle:

- If a driver is not connected to, the Driver Manager connects to the driver and calls SQLAllocHandle with a HandleType of SQL_HANDLE_ENV, SQLAllocHandle with a HandleType of SQL_HANDLE_DBC, SQLSetConnectAttr (if the application specified any connection attributes), and the connection function in the driver. The Driver Manager returns SQLSTATE IM006 (Driver's SQLSetConnectOption failed) and SQL_SUCCESS_WITH_INFO for the connection function if the driver returned an error for SQLSetConnectAttr. For more information, see Connecting to a Data Source or Driver.

- If the specified driver is already connected to on the ConnectionHandle, the Driver Manager calls only the connection function in the driver. In this case, the driver must make sure that all connection attributes for the ConnectionHandle maintain their current settings.

- If a different driver is connected to, the Driver Manager calls SQLFreeHandle with a HandleType of SQL_HANDLE_DBC, and then, if no other driver is connected to in that environment, it calls SQLFreeHandle with a HandleType of SQL_HANDLE_ENV in the connected driver and then disconnects that driver. It then performs the same operations as when a driver is not connected to.

The driver then allocates handles and initializes itself.

When the application calls SQLDisconnect, the Driver Manager calls SQLDisconnect in the driver. However, it does not disconnect the driver. This keeps the driver in memory for applications that repeatedly connect to and disconnect from a data source. When the application calls SQLFreeHandle with a HandleType of SQL_HANDLE_DBC, the Driver Manager calls SQLFreeHandle with a HandleType of SQL_HANDLE_DBC and then SQLFreeHandle with a HandleType of SQL_HANDLE_ENV in the driver, and then disconnects the driver.

An ODBC application can establish more than one connection.

Driver Manager Guidelines

The contents of *ServerName affect how the Driver Manager and a driver work together to establish a connection to a data source.

- If *ServerName contains a valid data source name, the Driver Manager locates the
corresponding data source specification in the system information and connects to the associated driver. The Driver Manager passes each SQLConnect argument to the driver.

- If the data source name cannot be found or ServerName is a null pointer, the Driver Manager locates the default data source specification and connects to the associated driver. The Driver Manager passes to the driver the UserName and Authentication arguments unmodified, and "DEFAULT" for the ServerName argument.

- If the ServerName argument is "DEFAULT", the Driver Manager locates the default data source specification and connects to the associated driver. The Driver Manager passes each SQLConnect argument to the driver.

- If the data source name cannot be found or ServerName is a null pointer, and the default data source specification does not exist, the Driver Manager returns SQL_ERROR with SQLSTATE IM002 (Data source name not found and no default driver specified).

After it is connected to by the Driver Manager, a driver can locate its corresponding data source specification in the system information and use driver-specific information from the specification to complete its set of required connection information.

If a default translation library is specified in the system information for the data source, the driver connects to it. A different translation library can be connected to by calling SQLSetConnectAttr with the SQL_ATTR_TRANSLATE_LIB attribute. A translation option can be specified by calling SQLSetConnectAttr with the SQL_ATTR_TRANSLATE_OPTION attribute.

If a driver supports SQLConnect, the driver keyword section of the system information for the driver must contain the ConnectFunctions keyword with the first character set to "Y."

**Connection Pooling**

Connection pooling allows an application to reuse a connection that has already been created. When connection pooling is enabled and SQLConnect is called, the Driver Manager tries to make the connection using a connection that is part of a pool of connections in an environment that has been designated for connection pooling. This environment is a shared environment that is used by all applications that use the connections in the pool.

Connection pooling is enabled before the environment is allocated by calling SQLSetEnvAttr to set SQL_ATTR_CONNECTION_POOLING to SQL_CP_ONE_PER_DRIVER (which specifies a maximum of one pool per driver) or SQL_CP_ONE_PER_HENV (which specifies a maximum of one pool per environment). SQLSetEnvAttr in this case is called with EnvironmentHandle set to null, which makes the attribute a process-level attribute. If SQL_ATTR_CONNECTION_POOLING is set to SQL_CP_OFF, connection pooling is disabled.

After connection pooling has been enabled, SQLAllocHandle with a HandleType of SQL_HANDLE_ENV is called to allocate an environment. The environment allocated by this call is a shared environment because connection pooling has been enabled. However, the environment that will be used is not determined until SQLAllocHandle with a HandleType of SQL_HANDLE_DBC is called.

SQLAllocHandle with a HandleType of SQL_HANDLE_DBC is called to allocate a connection. The Driver Manager tries to find an existing shared environment that matches the environment attributes set by the application. If no such environment exists, one is created as an implicit shared environment. If a matching shared environment is found, the environment handle is returned to the application and its reference count is incremented.

However, the connection that will be used is not determined until SQLConnect is called. At that
When the application calls SQLConnect to disconnect, the connection is returned to the connection pool and is available for reuse.

Optimizing Connection Pooling Performance

When distributed transactions are involved, it is possible to optimize connection pooling performance by using SQL_DTC_TRANSITION_COST, which is a SQLUINTEGER bitmask. The transitions referred to are the transitions of the connection attribute SQL_ATTR_ENLIST_IN_DTC going from value 0 to nonzero, and vice versa. This is a connection going from not enlisted in a distributed transaction to enlisted in a distributed transaction, and vice versa. Depending on how the driver has implemented enlistment (setting connection attribute SQL_ATTR_ENLIST_IN_DTC),
these transitions may be expensive and should therefore be avoided for best performance.

The value returned by the driver contains any combination of the following bits:

- **SQL_DTC_ENLIST_EXPENSIVE**, when set, implies the zero to nonzero transition is significantly more expensive than a transition from nonzero to another nonzero value (enlisting a previously enlisted connection in its next transaction).

- **SQL_DTC_UNENLIST_EXPENSIVE**, when set, implies the nonzero to zero transition is significantly more expensive than using a connection whose SQL_ATTR_ENLIST_IN_DTC attribute is already set to zero.

There is a performance versus connection usage tradeoff. If a driver indicates that one or more of these transitions are expensive, the driver manager's connection pooler responds to this by keeping more connections in the pool. Some of the connections in the pool are preferred for nontransactional use, and some are preferred for transactional use. However, if the driver indicates that these transitions are not expensive, fewer connections can be used, perhaps alternating between nontransactional and transactional use.

Drivers that do not support SQL_ATTR_ENLIST_IN_DTC do not need to support SQL_DTC_TRANSITION_COST. For drivers that support SQL_ATTR_ENLIST_IN_DTC but not SQL_DTC_TRANSITION_COST, it is assumed that the transitions are not expensive, as if the driver returned 0 (no bits set) for this value.

Although SQL_DTC_TRANSITION_COST was introduced in ODBC 3.5, an ODBC 2.x driver can also support it because the driver manager will query this information regardless of the driver version.

**Code Example**

In the following example, an application allocates environment and connection handles. It then connects to the SalesOrders data source with the user ID JohnS and the password Sesame and processes data. When it has finished processing data, it disconnects from the data source and frees the handles.

```c
#include <windows.h>
#include <sqlext.h>

int main() { 
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLHSTMT hstmt;
    SQLRETURN retcode;

    SQLCHAR * OutConnStr = (SQLCHAR *)malloc(255);
    SQLSMALLINT * OutConnStrLen = (SQLSMALLINT *)malloc(255);

    // Allocate environment handle
    retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

    // Set the ODBC version environment attribute
```
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
}

// Allocate connection handle
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
}

// Set login timeout to 5 seconds
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
}

// Connect to data source
retcode = SQLConnect(hdbc, (SQLCHAR*)"NorthWind", SQL_NTS, (SQLCHAR*)NULL);

// Allocate statement handle
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
}

// Process data
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
}

SQLDisconnect(hdbc);
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
}
}

SQLFreeHandle(SQL_HANDLE_ENV, henv);

---

### Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocating a handle</td>
<td>SQLAllocHandle Function</td>
</tr>
<tr>
<td>Discovering and enumerating values required to connect to a data source</td>
<td>SQLBrowseConnect Function</td>
</tr>
<tr>
<td>Disconnecting from a data source</td>
<td>SQLDisconnect Function</td>
</tr>
<tr>
<td>Connecting to a data source using a connection string or dialog box</td>
<td>SQLDriverConnect Function</td>
</tr>
<tr>
<td>Returning the setting of a connection attribute</td>
<td>SQLGetConnectAttr Function</td>
</tr>
<tr>
<td>Setting a connection attribute</td>
<td>SQLSetConnectAttr Function</td>
</tr>
</tbody>
</table>
SQLCopyDesc Function

Conformance
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary
SQLCopyDesc copies descriptor information from one descriptor handle to another.

Syntax

```
SQLRETURN SQLCopyDesc(
    SQLHDESC SourceDescHandle,
    SQLHDESC TargetDescHandle);
```

Arguments

SourceDescHandle
[Input] Source descriptor handle.

TargetDescHandle
[Input] Target descriptor handle. The TargetDescHandle argument can be a handle to an application descriptor or an IPD. TargetDescHandle cannot be set to a handle to an IRD, or SQLCopyDesc will return SQLSTATE HY016 (Cannot modify an implementation row descriptor).

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics
When `SQLCopyDesc` returns `SQL_ERROR` or `SQL_SUCCESS_WITH_INFO`, an associated `SQLSTATE` value can be obtained by calling `SQLGetDiagRec` with a `HandleType` of `SQL_HANDLE_DESC` and a `Handle` of `TargetDescHandle`. If an invalid `SourceDescHandle` was passed in the call, `SQL_INVALID_HANDLE` will be returned but no `SQLSTATE` will be returned. The following table lists the `SQLSTATE` values commonly returned by `SQLCopyDesc` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of `SQLSTATE`s returned by the Driver Manager. The return code associated with each `SQLSTATE` value is `SQL_ERROR`, unless noted otherwise.

When an error is returned, the call to `SQLCopyDesc` is immediately aborted, and the contents of the fields in the `TargetDescHandle` descriptor are undefined.

Because `SQLCopyDesc` may be implemented by calling `SQLGetDescField` and `SQLSetDescField`, `SQLCopyDesc` may return `SQLSTATE`s returned by `SQLGetDescField` or `SQLSetDescField`.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns <code>SQL_SUCCESS_WITH_INFO</code>.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific <code>SQLSTATE</code> and for which no implementation-specific <code>SQLSTATE</code> was defined. The error message returned by <code>SQLGetDiagRec</code> in the <code>*MessageText</code> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate the memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY007</td>
<td>Associated statement is not prepared</td>
<td><code>SourceDescHandle</code> was associated with an IRD, and the associated statement handle was not in the prepared or executed state.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) The descriptor handle in <code>SourceDescHandle</code> or <code>TargetDescHandle</code> was associated with a <code>StatementHandle</code> for which an asynchronously executing function (not this one) was called and was still executing when this function was called.</td>
</tr>
</tbody>
</table>

(DM) The descriptor handle in `SourceDescHandle` or `TargetDescHandle` was associated with a `StatementHandle` for which `SQLExecute`, `SQLExecDirect`, `SQLBulkOperations`, or `SQLSetPos` was called and returned `SQL_NEED_DATA`. This function was called before data was sent for all data-at-execution parameters or columns.

(DM) An asynchronously executing function was called for the connection handle that is associated with the `SourceDescHandle` or `TargetDescHandle`. This asynchronous function was still executing when the `SQLCopyDesc` function was called.

(DM) `SQLExecute`, `SQLExecDirect`, or `SQLMoreResults` was
called for one of the statement handles associated with the SourceDescHandle or TargetDescHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.

<table>
<thead>
<tr>
<th>Code</th>
<th>Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
</tr>
<tr>
<td>HY016</td>
<td>Cannot modify an implementation row descriptor</td>
</tr>
<tr>
<td>HY021</td>
<td>Inconsistent descriptor information</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute/option identifier</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
</tr>
</tbody>
</table>

**Comments**

A call to SQLCopyDesc copies the fields of the source descriptor handle to the target descriptor handle. Fields can be copied only to an application descriptor or an IPD, but not to an IRD. Fields can be copied from either an application or an implementation descriptor.

Fields can be copied from an IRD only if the statement handle is in the prepared or executed state; otherwise, the function returns SQLSTATE HY007 (Associated statement is not prepared).

Fields can be copied from an IPD whether or not a statement has been prepared. If an SQL statement with dynamic parameters has been prepared and automatic population of the IPD is supported and enabled, then the IPD is populated by the driver. When SQLCopyDesc is called with the IPD as the SourceDescHandle, the populated fields are copied. If the IPD is not populated by the driver, the contents of the fields originally in the IPD are copied.
All fields of the descriptor, except SQL_DESC_ALLOC_TYPE (which specifies whether the descriptor handle was automatically or explicitly allocated), are copied, whether or not the field is defined for the destination descriptor. Copied fields overwrite the existing fields.

The driver copies all descriptor fields if the SourceDescHandle and TargetDescHandle arguments are associated with the same driver, even if the drivers are on two different connections or environments. If the SourceDescHandle and TargetDescHandle arguments are associated with different drivers, the Driver Manager copies ODBC-defined fields, but does not copy driver-defined fields or fields that are not defined by ODBC for the type of descriptor.

The call to SQLCopyDesc is aborted immediately if an error occurs.

When the SQL_DESC_DATA_PTR field is copied, a consistency check is performed on the target descriptor. If the consistency check fails, SQLSTATE HY021 (Inconsistent descriptor information) is returned and the call to SQLCopyDesc is immediately aborted. For more information on consistency checks, see "Consistency Checks" in SQLSetDescRec Function.

Descriptor handles can be copied across connections even if the connections are under different environments. If the Driver Manager detects that the source and the destination descriptor handles do not belong to the same connection and the two connections belong to separate drivers, it implements SQLCopyDesc by performing a field-by-field copy using SQLGetDescField and SQLSetDescField.

When SQLCopyDesc is called with a SourceDescHandle on one driver and a TargetDescHandle on another driver, the error queue of the SourceDescHandle is cleared. This occurs because SQLCopyDesc in this case is implemented by calls to SQLGetDescField and SQLSetDescField.

An application might be able to associate an explicitly allocated descriptor handle with a StatementHandle, rather than calling SQLCopyDesc to copy fields from one descriptor to another. An explicitly allocated descriptor can be associated with another StatementHandle on the same ConnectionHandle by setting the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC statement attribute to the handle of the explicitly allocated descriptor. When this is done, SQLCopyDesc does not have to be called to copy descriptor field values from one descriptor to another. A descriptor handle cannot be associated with a StatementHandle on another ConnectionHandle, however; to use the same descriptor field values on StatementHandles on different ConnectionHandles, SQLCopyDesc has to be called.

For a description of the fields in a descriptor header or record, see SQLSetDescField Function. For more information on descriptors, see Descriptors.

### Copying Rows Between Tables

An application may copy data from one table to another without copying the data at the application level. To do this, the application binds the same data buffers and descriptor information to a statement that fetches the data and the statement that inserts the data into a copy. This can be accomplished either by sharing an application descriptor (binding an explicitly allocated descriptor as both the ARD to one statement and the APD in another) or by using SQLCopyDesc to copy the bindings between the ARD and the APD of the two statements. If the statements are on different connections, SQLCopyDesc must be used. In addition, SQLCopyDesc has to be called to copy the bindings between the IRD and the IPD of the two statements. When copying across statements on the same connection, the SQL_ACTIVE_STATEMENTS information type returned by the driver for a call to SQLGetInfo must be greater than 1 for this operation to succeed. (This is not the case when copying across connections.)
Code Example

In the following example, descriptor operations are used to copy the fields of the PartsSource table into the PartsCopy table. The contents of the PartsSource table are fetched into rowset buffers in hstmt0. These values are used as parameters of an INSERT statement on hstmt1 to populate the columns of the PartsCopy table. To do so, the fields of the IRD of hstmt0 are copied to the fields of the IPD of hstmt1, and the fields of the ARD of hstmt0 are copied to the fields of the APD of hstmt1. Use SQLSetDescField to set the IPD’S SQL_DESC_PARAMETER_TYPE attribute to SQL_PARAM_INPUT when you copy IRD fields from a statement with output parameters to IPD fields that need to be input parameters.

```c
#define ROWS 100
#define DESC_LEN 50
#define SQL_SUCCEEDED(rc) (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)

// Template for a row
typedef struct {
    SQLINTEGER sPartID;
    SQLINTEGER cbPartID;
    SQLUCHAR szDescription[DESC_LENGTH];
    SQLINTEGER cbDescription;
    REAL sPrice;
    SQLINTEGER cbPrice;
} PartsSource;

PartsSource rget[ROWS]; // rowset buffer
SQLUSMALLINT sts_ptr[ROWS]; // status pointer
SQLHSTMT hstmt0, hstmt1;
SQLHDESC hArd0, hIrd0, hApd1, hIpd1;

// ARD and IRD of hstmt0
SQLGetStmtAttr(hstmt0, SQL_ATTR_APP_ROW_DESC, &hArd0, 0, NULL);
SQLGetStmtAttr(hstmt0, SQL_ATTR_IMP_ROW_DESC, &hIrd0, 0, NULL);

// APD and IPD of hstmt1
SQLGetStmtAttr(hstmt1, SQL_ATTR_APP_PARAM_DESC, &hApd1, 0, NULL);
SQLGetStmtAttr(hstmt1, SQL_ATTR_IMP_PARAM_DESC, &hIpd1, 0, NULL);

// Use row-wise binding on hstmt0 to fetch rows
SQLSetStmtAttr(hstmt0, SQL_ATTR_ROW_BIND_TYPE, (SQLPOINTER) sizeof(PartsSource), 0);

// Set rowset size for hstmt0
SQLSetStmtAttr(hstmt0, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER) ROWS, 0);

// Execute a select statement
SQLExecDirect(hstmt0, "SELECT PARTID, DESCRIPTION, PRICE FROM PARTS ORDER BY 3, 1, 2" SQL_NTS);

// Bind
SQLBindCol(hstmt0, 1, SQL_C_SLONG, rget[0].sPartID, 0, &rget[0].cbPartID);
SQLBindCol(hstmt0, 2, SQL_C_CHAR, &rget[0].szDescription, DESC_LEN, &rget[0].cbDescription);
SQLBindCol(hstmt0, 3, SQL_C_FLOAT, &rget[0].sPrice, 0, &rget[0].cbPrice);
```
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting multiple descriptor fields</td>
<td>SQLGetDescRec Function</td>
</tr>
<tr>
<td>Setting a single descriptor field</td>
<td>SQLSetDescField Function</td>
</tr>
<tr>
<td>Setting multiple descriptor fields</td>
<td>SQLSetDescRec Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files
SQLDataSources Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary

SQLDataSources returns information about a data source. This function is implemented only by the Driver Manager.

Syntax

```c
SQLRETURN SQLDataSources(
    SQLHENV EnvironmentHandle,
    SQLUSMALLINT Direction,
    SQLCHAR * ServerName,
    SQLSMALLINT BufferLength1,
    SQLSMALLINT * NameLength1Ptr,
    SQLCHAR * Description,
    SQLSMALLINT BufferLength2,
    SQLSMALLINT * NameLength2Ptr);
```

Arguments

EnvironmentHandle
[Input] Environment handle.

Direction
[Input] Determines which data source the Driver Manager returns information about. Can be:

SQL_FETCH_NEXT (to fetch the next data source name in the list), SQL_FETCH_FIRST (to fetch from the beginning of the list), SQL_FETCH.First_USER (to fetch the first user DSN), or SQL_FETCH.First_SYSTEM (to fetch the first system DSN).

When Direction is set to SQL_FETCH_FIRST, subsequent calls to SQLDataSources with Direction set to SQL_FETCH_NEXT return both user and system DSNs. When Direction is set to SQL_FETCH.First_USER, all subsequent calls to SQLDataSources with Direction set to SQL_FETCH.Next return only user DSNs. When Direction is set to SQL_FETCH.First_SYSTEM, all subsequent calls to SQLDataSources with Direction set to SQL_FETCH.Next return only system DSNs.

ServerName
[Output] Pointer to a buffer in which to return the data source name.

If ServerName is NULL, NameLength1Ptr will still return the total number of characters
BufferLength1
[Input] Length of the *ServerName buffer, in characters; this does not need to be longer than SQL_MAX_DSN_LENGTH plus the null-termination character.

NameLength1Ptr
[Output] Pointer to a buffer in which to return the total number of characters (excluding the null-termination character) available to return in *ServerName. If the number of characters available to return is greater than or equal to BufferLength1, the data source name in *ServerName is truncated to BufferLength1 minus the length of a null-termination character.

Description
[Output] Pointer to a buffer in which to return the description of the driver associated with the data source. For example, dBASE or SQL Server.

If Description is NULL, NameLength2Ptr will still return the total number of characters (excluding the null-termination character for character data) available to return in the buffer pointed to by Description.

BufferLength2
[Input] Length in characters of the *Description buffer.

NameLength2Ptr
[Output] Pointer to a buffer in which to return the total number of characters (excluding the null-termination character) available to return in *Description. If the number of characters available to return is greater than or equal to BufferLength2, the driver description in *Description is truncated to BufferLength2 minus the length of a null-termination character.

Returns
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics
When SQLDataSources returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_ENV and a Handle of EnvironmentHandle. The following table lists the SQLSTATE values typically returned by SQLDataSources and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>(DM) Driver Manager–specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right</td>
<td>(DM) The buffer *ServerName was not large enough to return the complete data source name. Therefore, the</td>
</tr>
</tbody>
</table>
name was truncated. The length of the entire data source name is returned in *NameLength1Ptr. (Function returns SQL_SUCCESS_WITH_INFO.)

(DM) The buffer *Description was not large enough to return the complete driver description. Therefore, the description was truncated. The length of the untruncated data source description is returned in *NameLength2Ptr. (Function returns SQL_SUCCESS_WITH_INFO.)

<table>
<thead>
<tr>
<th>HY000</th>
<th>General error</th>
<th>(DM) An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>(DM) The Driver Manager was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
</tbody>
</table>
| HY090   | Invalid string or buffer length | (DM) The value specified for argument BufferLength1 was less than 0.  
(DM) The value specified for argument BufferLength2 was less than 0. |
| HY103   | Invalid retrieval code | (DM) The value specified for the argument Direction was not equal to SQL_FETCH_FIRST, SQL_FETCH_FIRST_USER, SQL_FETCH_FIRST_SYSTEM, or SQL_FETCH_NEXT.                                                                                           |
| HY117   | Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.                                           | (DM) For more information about suspended state, see SQLEndTran Function.                                                                                                                                                                                   |

Comments

Because SQLDataSources is implemented in the Driver Manager, it is supported for all drivers regardless of a particular driver's standards compliance.

An application can call SQLDataSources multiple times to retrieve all data source names. The Driver
Manager retrieves this information from the system information. When there are no more data source names, the Driver Manager returns SQL_NO_DATA. If SQLDataSources is called with SQL_FETCH_NEXT immediately after it returns SQL_NO_DATA, it will return the first data source name. For information about how an application uses the information returned by SQLDataSources, see Choosing a Data Source or Driver.

If SQL_FETCH_NEXT is passed to SQLDataSources the very first time it is called, it will return the first data source name.

The driver determines how data source names are mapped to actual data sources.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovering and listing values required to connect to a data source</td>
<td>SQLBrowseConnect Function</td>
</tr>
<tr>
<td>Connecting to a data source</td>
<td>SQLConnect Function</td>
</tr>
<tr>
<td>Connecting to a data source using a connection string or dialog box</td>
<td>SQLDriverConnect Function</td>
</tr>
<tr>
<td>Returning driver descriptions and attributes</td>
<td>SQLDrivers Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLDescribeCol Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLDescribeCol returns the result descriptor — column name, type, column size, decimal digits, and nullability — for one column in the result set. This information also is available in the fields of the IRD.

Syntax
Arguments

StatementHandle
[Input] Statement handle.

ColumnNumber
[Input] Column number of result data, ordered sequentially in increasing column order, starting at 1. The ColumnNumber argument can also be set to 0 to describe the bookmark column.

ColumnName
[Output] Pointer to a null-terminated buffer in which to return the column name. This value is read from the SQL_DESC_NAME field of the IRD. If the column is unnamed or the column name cannot be determined, the driver returns an empty string.

If ColumnName is NULL, NameLengthPtr will still return the total number of characters (excluding the null-termination character for character data) available to return in the buffer pointed to by ColumnName.

BufferLength
[Input] Length of the *ColumnName buffer, in characters.

NameLengthPtr
[Output] Pointer to a buffer in which to return the total number of characters (excluding the null termination) available to return in *ColumnName. If the number of characters available to return is greater than or equal to BufferLength, the column name in *ColumnName is truncated to BufferLength minus the length of a null-termination character.

DataTypePtr
[Output] Pointer to a buffer in which to return the SQL data type of the column. This value is read from the SQL_DESC_CONCISE_TYPE field of the IRD. This will be one of the values in SQL Data Types, or a driver-specific SQL data type. If the data type cannot be determined, the driver returns SQL_UNKNOWN_TYPE.

In ODBC 3.x, SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP is returned in *DataTypePtr for date, time, or timestamp data, respectively; in ODBC 2.x, SQL_DATE, SQL_TIME, or SQL_TIMESTAMP is returned. The Driver Manager performs the required mappings when an ODBC 2.x application is working with an ODBC 3.x driver or when an ODBC 3.x application is working with an ODBC 2.x driver.

When ColumnNumber is equal to 0 (for a bookmark column), SQL_BINARY is returned in *DataTypePtr for variable-length bookmarks. (SQL_INTEGER is returned if bookmarks are used by an ODBC 3.x application working with an ODBC 2.x driver or by an ODBC 2.x application working with an ODBC 3.x driver.)

SQLRETURN SQLDescribeCol(
    SQLHSTMT StatementHandle,
    SQLUSMALLINT ColumnNumber,
    SQLCHAR * ColumnName,
    SQLSMALLINT BufferLength,
    SQLSMALLINT * NameLengthPtr,
    SQLSMALLINT * DataTypePtr,
    SQULENUM * ColumnSizePtr,
    SQLSMALLINT * DecimalDigitsPtr,
    SQLSMALLINT * NullablePtr);
For more information on these data types, see SQL Data Types in Appendix D: Data Types. For information about driver-specific SQL data types, see the driver’s documentation.

**ColumnSizePtr**
[Output] Pointer to a buffer in which to return the size (in characters) of the column on the data source. If the column size cannot be determined, the driver returns 0. For more information on column size, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

**DecimalDigitsPtr**
[Output] Pointer to a buffer in which to return the number of decimal digits of the column on the data source. If the number of decimal digits cannot be determined or is not applicable, the driver returns 0. For more information on decimal digits, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.

**NullablePtr**
[Output] Pointer to a buffer in which to return a value that indicates whether the column allows NULL values. This value is read from the SQL_DESC_NULLABLE field of the IRD. The value is one of the following:

- SQL_NO_NULLS: The column does not allow NULL values.
- SQL_NULLABLE: The column allows NULL values.
- SQL_NULLABLE_UNKNOWN: The driver cannot determine if the column allows NULL values.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When SQLDescribeCol returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLDescribeCol and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The buffer *ColumnName was not large enough to return the entire column name, so the column name was truncated. The length of the untruncated column name is returned in *NameLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Detailed Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>07005</td>
<td>Prepared statement not a cursor-specification</td>
<td>The statement associated with the <code>StatementHandle</code> did not return a result set. There were no columns to describe.</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>(DM) The value specified for the argument <code>ColumnNumber</code> was equal to 0, and the SQL_ATTR_USE_BOOKMARKS statement option was SQL_UB_OFF. The value specified for the argument <code>ColumnNumber</code> was greater than the number of columns in the result set.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation failure</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <code>StatementHandle</code>. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the <code>StatementHandle</code>. Then the function was called again on the <code>StatementHandle</code>. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the <code>StatementHandle</code> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when SQLDescribeCol was called.</td>
</tr>
</tbody>
</table>

(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the `StatementHandle` and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.

(DM) An asynchronously executing function (not this one) was called for the `StatementHandle` and was still executing when this function was called.

(DM) The function was called prior to calling SQLPrepare, SQLExecute, or a catalog function on the statement handle.

(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the
**StatementHandle** and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value specified for argument BufferLength was less than 0.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td>SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

**SQLDescribeCol** can return any SQLSTATE that can be returned by **SQLPrepare** or **SQLExecute** when called after **SQLPrepare** and before **SQLExecute**, depending on when the data source evaluates the SQL statement associated with the statement handle.

For performance reasons, an application should not call **SQLDescribeCol** before executing a statement.

**Comments**

An application typically calls **SQLDescribeCol** after a call to **SQLPrepare** and before or after the associated call to **SQLExecute**. An application can also call **SQLDescribeCol** after a call to **SQLExecDirect**. For more information, see Result Set Metadata.

**SQLDescribeCol** retrieves the column name, type, and length generated by a **SELECT** statement. If the column is an expression, *ColumnName* is either an empty string or a driver-defined name.
**Note**

ODBC supports SQL_NULLABLE_UNKNOWN as an extension, even though the Open Group and SQL Access Group Call Level Interface specification does not specify the option for `SQLDescribeCol`.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td><code>SQLBindCol</code></td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td><code>SQLCancel</code></td>
</tr>
<tr>
<td>Returning information about a column in a result set</td>
<td><code>SQLColAttribute</code></td>
</tr>
<tr>
<td>Fetching multiple rows of data</td>
<td><code>SQLFetch</code></td>
</tr>
<tr>
<td>Returning the number of result set columns</td>
<td><code>SQLNumResultCols</code></td>
</tr>
<tr>
<td>Preparing a statement for execution</td>
<td><code>SQLPrepare</code></td>
</tr>
</tbody>
</table>

**See Also**

[ODBC API Reference](#)

[ODBC Header Files](#)

© 2016 Microsoft

**SQLDescribeParam Function**

**Conformance**

Version Introduced: ODBC 1.0 Standards Compliance: ODBC

**Summary**

`SQLDescribeParam` returns the description of a parameter marker associated with a prepared SQL statement. This information is also available in the fields of the IPD.

**Syntax**

```c
SQLRETURN SQLDescribeParam(
    SQLHSTMT StatementHandle,
```
Argument

StatementHandle

[Input] Statement handle.

ParameterNumber

[Input] Parameter marker number ordered sequentially in increasing parameter order, starting at 1.

DataTypePtr

[Output] Pointer to a buffer in which to return the SQL data type of the parameter. This value is read from the SQL_DESC_CONCISE_TYPE record field of the IPD. This will be one of the values in the SQL Data Types section of Appendix D: Data Types, or a driver-specific SQL data type.

In ODBC 3.x, SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP will be returned in *DataTypePtr for date, time, or timestamp data, respectively; in ODBC 2.x, SQL_DATE, SQL_TIME, or SQL_TIMESTAMP will be returned. The Driver Manager performs the required mappings when an ODBC 2.x application is working with an ODBC 3.x driver or when an ODBC 3.x application is working with an ODBC 2.x driver.

When ColumnNumber is equal to 0 (for a bookmark column), SQL_BINARY is returned in *DataTypePtr for variable-length bookmarks. (SQL_INTEGER is returned if bookmarks are used by an ODBC 3.x application working with an ODBC 2.x driver or by an ODBC 2.x application working with an ODBC 3.x driver.)

For more information, see SQL Data Types in Appendix D: Data Types. For information about driver-specific SQL data types, see the driver's documentation.

ParameterSizePtr

[Output] Pointer to a buffer in which to return the size, in characters, of the column or expression of the corresponding parameter marker as defined by the data source. For more information about column size, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size.

DecimalDigitsPtr

[Output] Pointer to a buffer in which to return the number of decimal digits of the column or expression of the corresponding parameter as defined by the data source. For more information about decimal digits, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size.

NullablePtr

[Output] Pointer to a buffer in which to return a value that indicates whether the parameter allows NULL values. This value is read from the SQL_DESC_NULLABLE field of the IPD. One of the following:

- SQL_NO_NULLS: The parameter does not allow NULL values (this is the default value).
- SQL_NULLABLE: The parameter allows NULL values.
- SQL_NULLABLE_UNKNOWN: The driver cannot determine whether the parameter allows NULL values.
Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLDescribeParam returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values typically returned by SQLDescribeParam and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>(DM) The value specified for the argument ParameterNumber is less than 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value specified for the argument ParameterNumber was greater than the number of parameters in the associated SQL statement.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parameter marker was part of a non-DML statement.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parameter marker was part of a SELECT list.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>21S01</td>
<td>Insert value list does not match column list</td>
<td>The number of parameters in the INSERT statement did not match the number of columns in the table named in the statement.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code>. Then the function was called again on the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) The function was called before calling <code>SQLPrepare</code> or <code>SQLExecDirect</code> for the <code>StatementHandle</code>. (DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when the <code>SQLDescribeParam</code> function was called. (DM) An asynchronously executing function (not this one) was called for the <code>StatementHandle</code> and was still executing when this function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for the <code>StatementHandle</code> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>, <code>SQL_ATTR_CONNECTION_TIMEOUT</code>.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the <code>StatementHandle</code> does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><code>SQLCompleteAsync</code> has not been called to</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is...</td>
</tr>
</tbody>
</table>
complete the previous asynchronous operation on this handle.

enabled, **SQLCompleteAsync** must be called on the handle to do post-processing and complete the operation.

Comments

Parameter markers are numbered in increasing parameter order, starting with 1, in the order they appear in the SQL statement.

**SQLDescribeParam** does not return the type (input, input/output, or output) of a parameter in an SQL statement. Except in calls to procedures, all parameters in SQL statements are input parameters. To determine the type of each parameter in a call to a procedure, an application calls **SQLProcedureColumns**.

For more information, see Describing Parameters.

Code Example

The following example prompts the user for an SQL statement and then prepares that statement. Next, it calls **SQLNumParams** to determine whether the statement contains any parameters. If the statement contains parameters, it calls **SQLDescribeParam** to describe those parameters and **SQLBindParameter** to bind them. Finally, it prompts the user for the values of any parameters and then executes the statement.

```c
SQLCHAR Statement[100];
SQLSMALLINT NumParams, i, DataType, DecimalDigits, Nullable;
SQLINTEGER ParamSize;
SQLHSTMT hstmt;

// Prompt the user for an SQL statement and prepare it.
GetSQLStatement(Statement);
SQLPrepare(hstmt, Statement, SQL_NTS);

// Check to see if there are any parameters. If so, process them.
SQLNumParams(hstmt, &NumParams);
if (NumParams) {
   // Allocate memory for three arrays. The first holds pointers to buffers in which
   // each parameter value will be stored in character form. The second contains the
   // length of each buffer. The third contains the length/indicator value for each
   // parameter.
   SQLPOINTER * PtrArray = (SQLPOINTER *) malloc(NumParams * sizeof(SQLPOINTER));
   SQLINTEGER * BufferLenArray = (SQLINTEGER *) malloc(NumParams * sizeof(SQLINTEGER));
   SQLINTEGER * LenOrIndArray = (SQLINTEGER *) malloc(NumParams * sizeof(SQLINTEGER));

   for (i = 0; i < NumParams; i++) {
      // Describe the parameter.
      SQLDescribeParam(hstmt, i + 1, &DataType, &ParamSize, &DecimalDigits, &Nullable);

      // Call a helper function to allocate a buffer in which to store the parameter
      // value in character form. The function determines the size of the buffer from
```
// the SQL data type and parameter size returned by SQLDescribeParam and returns
// a pointer to the buffer and the length of the buffer.
AllocParamBuffer(DataType, ParamSize, &PtrArray[i], &BufferLenArray[i]);

// Bind the memory to the parameter. Assume that we only have input parameters.
SQLBindParameter(hstmt, i + 1, SQL_PARAM_INPUT, SQL_C_CHAR, DataType, ParamSize,
                 DecimalDigits, PtrArray[i], BufferLenArray[i],
                 &LenOrIndArray[i]);

// Prompt the user for the value of the parameter and store it in the memory
// allocated earlier. For simplicity, this function does not check the value
// against the information returned by SQLDescribeParam. Instead, the driver does
// this when the statement is executed.
GetParamValue(PtrArray[i], BufferLenArray[i], &LenOrIndArray[i]);
}
}

// Execute the statement.
SQLExecute(hstmt);

// Process the statement further, such as retrieving results (if any) and closing the
// cursor (if any). Code not shown.

// Free the memory allocated for each parameter and the memory allocated for the arrays
// of pointers, buffer lengths, and length/indicator values.
for (i = 0; i < NumParams; i++) free(PtrArray[i]);
free(PtrArray);
free(BufferLenArray);
free(LenOrIndArray);

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a parameter</td>
<td>SQLBindParameter Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
<tr>
<td>Preparing a statement for execution</td>
<td>SQLPrepare Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft
SQLDisconnect Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLDisconnect closes the connection associated with a specific connection handle.

Syntax

```c
SQLRETURN SQLDisconnect(
    SQLHDBC ConnectionHandle);
```

Arguments

ConnectionHandle
  [Input] Connection handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_STILL_EXECUTING.

Diagnostics

When SQLDisconnect returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_DBC and a Handle of ConnectionHandle. The following table lists the SQLSTATE values commonly returned by SQLDisconnect and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01002</td>
<td>Disconnect error</td>
<td>An error occurred during the disconnect. However, the</td>
</tr>
<tr>
<td>Code</td>
<td>Error Description</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open (DM) The connection specified in the argument ConnectionHandle was not open.</td>
<td></td>
</tr>
<tr>
<td>25000</td>
<td>Invalid transaction state There was a transaction in progress on the connection specified by the argument ConnectionHandle. The transaction remains active.</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>General error An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
<td></td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error The driver was unable to allocate memory required to support execution or completion of the function.</td>
<td></td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled Asynchronous processing was enabled for the ConnectionHandle. The function was called, and before it finished executing SQLCancelHandle Function was called on the ConnectionHandle. Then the function was called again on the ConnectionHandle. The function was called, and before it finished executing SQLCancelHandle was called on the ConnectionHandle from a different thread in a multithread application.</td>
<td></td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error (DM) An asynchronously executing function was called for a StatementHandle associated with the ConnectionHandle and was still executing when SQLDisconnect was called. (DM) An asynchronously executing function (not this one) was called for the ConnectionHandle and was still executing when this function was called. (DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for a StatementHandle associated with the ConnectionHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
<td></td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
<td></td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed. (DM) For more information about suspended state, see SQLEndTran Function.</td>
<td></td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout The connection timeout period expired before the data</td>
<td></td>
</tr>
</tbody>
</table>
source responded to the request, and the connection is still active. The connection timeout period is set through `SQLSetConnectAttr`, `SQL_ATTR_CONNECTION_TIMEOUT`.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM001</td>
<td>Driver does not support this function (DM) The driver associated with the <code>ConnectionHandle</code> does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><code>SQLCompleteAsync</code> has not been called to complete the previous asynchronous operation on this handle. If the previous function call on the handle returns <code>SQL_STILL_EXECUTING</code> and if notification mode is enabled, <code>SQLCompleteAsync</code> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

**Comments**

If an application calls `SQLDisconnect` after `SQLBrowseConnect` returns SQL_NEED_DATA and before it returns a different return code, the driver cancels the connection browsing process and returns the connection to an unconnected state.

If an application calls `SQLDisconnect` while there is an incomplete transaction associated with the connection handle, the driver returns SQLSTATE 25000 (Invalid transaction state), indicating that the transaction is unchanged and the connection is open. An incomplete transaction is one that has not been committed or rolled back with `SQLEndTran`.

If an application calls `SQLDisconnect` before it has freed all statements associated with the connection, the driver, after it successfully disconnects from the data source, frees those statements and all descriptors that have been explicitly allocated on the connection. However, if one or more of the statements associated with the connection are still executing asynchronously, `SQLDisconnect` returns SQL_ERROR with a SQLSTATE value of HY010 (Function sequence error). Also, `SQLDisconnect` will free all associated statements and all descriptors that have been explicitly allocated on the connection, if the connection is in a suspended state or if `SQLDisconnect` was successfully canceled by `SQLCancelHandle`.

For information about how an application uses `SQLDisconnect`, see [Disconnecting from a Data Source or Driver](#).

**Disconnecting from a Pooled Connection**

If connection pooling is enabled for a shared environment and an application calls `SQLDisconnect` on a connection in that environment, the connection is returned to the connection pool and is still available to other components using the same shared environment.

**Code Example**
See Sample ODBC Program, SQLBrowseConnect Function, and SQLConnect Function.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocating a handle</td>
<td>SQLAllocHandle Function</td>
</tr>
<tr>
<td>Connecting to a data source</td>
<td>SQLConnect Function</td>
</tr>
<tr>
<td>Connecting to a data source using a connection string or dialog box</td>
<td>SQLDriverConnect Function</td>
</tr>
<tr>
<td>Executing a commit or rollback operation</td>
<td>SQLEndTran Function</td>
</tr>
<tr>
<td>Freeing a connection handle</td>
<td>SQLFreeConnect Function</td>
</tr>
</tbody>
</table>

See Also

- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft

SQLDriverConnect Function

**Conformance**

Version Introduced: ODBC 1.0 Standards Compliance: ODBC

**Summary**

SQLDriverConnect is an alternative to SQLConnect. It supports data sources that require more connection information than the three arguments in SQLConnect, dialog boxes to prompt the user for all connection information, and data sources that are not defined in the system information.

SQLDriverConnect provides the following connection attributes:

- Establish a connection using a connection string that contains the data source name, one or more user IDs, one or more passwords, and other information required by the data source.
- Establish a connection using a partial connection string or no additional information; in this case, the Driver Manager and the driver can each prompt the user for connection information.
- Establish a connection to a data source that is not defined in the system information. If the application supplies a partial connection string, the driver can prompt the user for
connection information.

- Establish a connection to a data source using a connection string constructed from the information in a .dsn file.

After a connection is established, **SQLDriverConnect** returns the completed connection string. The application can use this string for subsequent connection requests. For more information, see Connecting with SQLDriverConnect.

**Syntax**

```c
SQLRETURN SQLDriverConnect(
    SQLHDBC      ConnectionHandle,
    SQLHWND      WindowHandle,
    SQLCHAR *    InConnectionString,
    SQLSMALLINT  StringLength1,
    SQLCHAR *    OutConnectionString,
    SQLSMALLINT  BufferLength,
    SQLSMALLINT * StringLength2Ptr,
    SQLUSMALLINT DriverCompletion);
```

**Arguments**

**ConnectionHandle**

[Input] Connection handle.

**WindowHandle**

[Input] Window handle. The application can pass the handle of the parent window, if applicable, or a null pointer if either the window handle is not applicable or **SQLDriverConnect** will not present any dialog boxes.

**InConnectionString**

[Input] A full connection string (see the syntax in "Comments"), a partial connection string, or an empty string.

**StringLength1**

[Input] Length of *InConnectionString*, in characters if the string is Unicode, or bytes if string is ANSI or DBCS.

**OutConnectionString**

[Output] Pointer to a buffer for the completed connection string. Upon successful connection to the target data source, this buffer contains the completed connection string. Applications should allocate at least 1,024 characters for this buffer.

If **OutConnectionString** is NULL, **StringLength2Ptr** will still return the total number of characters (excluding the null-termination character for character data) available to return in the buffer pointed to by **OutConnectionString**.

**BufferLength**

[Input] Length of the *OutConnectionString* buffer, in characters.
**StringLength2Ptr**

[Output] Pointer to a buffer in which to return the total number of characters (excluding the null-termination character) available to return in *OutConnectionString*. If the number of characters available to return is greater than or equal to *BufferLength*, the completed connection string in *OutConnectionString* is truncated to *BufferLength* minus the length of a null-termination character.

**DriverCompletion**

[Input] Flag that indicates whether the Driver Manager or driver must prompt for more connection information:

- SQL_DRIVER_PROMPT
- SQL_DRIVER_COMPLETE
- SQL_DRIVER_COMPLETE_REQUIRED
- SQL_DRIVER_NOPROMPT

(For additional information, see "Comments.")

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_STILL_EXECUTING.

**Diagnostics**

When **SQLDriverConnect** returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling **SQLGetDiagRec** with an *fHandleType* of SQL_HANDLE_DBC and an *hHandle* of ConnectionHandle. The following table lists the SQLSTATE values commonly returned by **SQLDriverConnect** and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The buffer <em>OutConnectionString</em> was not large enough to return the entire connection string, so the connection string was truncated. The length of the untruncated connection string is returned in <em>StringLength2Ptr</em>. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S00</td>
<td>Invalid connection string attribute</td>
<td>An invalid attribute keyword was specified in the connection string (<em>InConnectionString</em>), but the driver was able to connect to the data source anyway. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Error Message</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed</td>
<td>The driver did not support the specified value pointed to by the <code>ValuePtr</code> argument in <code>SQLSetConnectAttr</code> and substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S08</td>
<td>Error saving file DSN</td>
<td>The string in <code>*InConnectionString</code> contained a <strong>FILEDSN</strong> keyword, but the .dsn file was not saved. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S09</td>
<td>Invalid keyword</td>
<td>(DM) The string in <code>*InConnectionString</code> contained a <strong>SAVEFILE</strong> keyword but not a <strong>DRIVER</strong> or a <strong>FILEDSN</strong> keyword. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08001</td>
<td>Client unable to establish connection</td>
<td>The driver was unable to establish a connection with the data source.</td>
</tr>
<tr>
<td>08002</td>
<td>Connection name in use</td>
<td>(DM) The specified <code>ConnectionHandle</code> had already been used to establish a connection with a data source, and the connection was still open.</td>
</tr>
<tr>
<td>08004</td>
<td>Server rejected the connection</td>
<td>The data source rejected the establishment of the connection for implementation-defined reasons.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was attempting to connect failed before the <code>SQLDriverConnect</code> function completed processing.</td>
</tr>
<tr>
<td>28000</td>
<td>Invalid authorization specification</td>
<td>Either the user identifier or the authorization string, or both, as specified in the connection string (<code>InConnectionString</code>), violated restrictions defined by the data source.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <code>*szMessageText</code> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error: Invalid file dsn</td>
<td>(DM) The string in <code>*InConnectionString</code> contained a FILEDSN keyword, but the name of the .dsn file was not found.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error: Unable to create file buffer</td>
<td>(DM) The string in <code>*InConnectionString</code> contained a FILEDSN keyword, but the .dsn file was unreadable.</td>
</tr>
<tr>
<td>Code</td>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The Driver Manager was unable to allocate memory required to support execution or completion of the <strong>SQLDriverConnect</strong> function. The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <strong>ConnectionHandle</strong>. The function was called, and before it completed execution, the <strong>SQLCancelHandle</strong> function was called on the <strong>ConnectionHandle</strong>, and then the <strong>SQLDriverConnect</strong> function was called again on the <strong>ConnectionHandle</strong>. Or, the <strong>SQLDriverConnect</strong> function was called, and before it completed execution, <strong>SQLCancelHandle</strong> was called on the <strong>ConnectionHandle</strong> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) Another asynchronously executing function (not <strong>SQLDriverConnect</strong>) was called for the <strong>ConnectionHandle</strong> and was still executing when the <strong>SQLDriverConnect</strong> function was called.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The <strong>SQLDriverConnect</strong> function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value specified for argument <strong>StringLength1</strong> was less than 0 and was not equal to SQL_NTS. (DM) The value specified for argument <strong>BufferLength</strong> was less than 0.</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute/option identifier</td>
<td>(DM) The <strong>DriverCompletion</strong> argument was SQL_DRIVER_PROMPT, and the <strong>WindowHandle</strong> argument was a null pointer.</td>
</tr>
<tr>
<td>HY110</td>
<td>Invalid driver completion</td>
<td>(DM) The value specified for the argument <strong>DriverCompletion</strong> was not equal to SQL_DRIVER_PROMPT, SQL_DRIVER_COMPLETE, SQL_DRIVER_COMPLETE_REQUIRED, or SQL_DRIVER_NOPROMPT. (DM) Connection pooling was enabled, and the value specified for the argument <strong>DriverCompletion</strong> was not equal to SQL_DRIVER_NOPROMPT.</td>
</tr>
<tr>
<td>Code</td>
<td>Condition Description</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The driver does not support the version of ODBC behavior that the application requested.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The login timeout period expired before the connection to the data source completed. The timeout period is set through SQLSetConnectAttr, SQL_ATTR_LOGIN_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver corresponding to the specified data source name does not support the function.</td>
</tr>
<tr>
<td>IM002</td>
<td>Data source not found and no default driver specified</td>
<td>(DM) The data source name specified in the connection string (InConnectionString) was not found in the system information, and there was no default driver specification. (DM) ODBC data source and default driver information could not be found in the system information.</td>
</tr>
<tr>
<td>IM003</td>
<td>Specified driver could not be loaded</td>
<td>(DM) The driver listed in the data source specification in the system information or specified by the DRIVER keyword was not found or could not be loaded for some other reason.</td>
</tr>
<tr>
<td>IM004</td>
<td>Driver's SQLAllocHandle on SQL_HANDLE_ENV failed</td>
<td>(DM) During SQLDriverConnect, the Driver Manager called the driver's SQLAllocHandle function with an fHandleType of SQL_HANDLE_ENV and the driver returned an error.</td>
</tr>
<tr>
<td>IM005</td>
<td>Driver's SQLAllocHandle on SQL_HANDLE_DBC failed</td>
<td>(DM) During SQLDriverConnect, the Driver Manager called the driver's SQLAllocHandle function with an fHandleType of SQL_HANDLE_DBC and the driver returned an error.</td>
</tr>
<tr>
<td>IM006</td>
<td>Driver's SQLSetConnectAttr failed</td>
<td>(DM) During SQLDriverConnect, the Driver Manager called the driver's SQLSetConnectAttr function and the driver returned an error.</td>
</tr>
<tr>
<td>IM007</td>
<td>No data source or driver specified; dialog prohibited</td>
<td>No data source name or driver was specified in the connection string, and DriverCompletion was SQL_DRIVER_NOPROMPT.</td>
</tr>
</tbody>
</table>
| IM008   | Dialog failed | The driver attempted to display its login dialog box and failed.  
  *WindowHandle* was a null pointer, and *DriverCompletion* was not SQL_DRIVER_NO_PROMPT. |
| IM009   | Unable to load translation DLL | The driver was unable to load the translation DLL that was specified for the data source or for the connection. |
| IM010   | Data source name too long | (DM) The attribute value for the DSN keyword was longer than SQL_MAX_DSN_LENGTH characters. |
| IM011   | Driver name too long | (DM) The attribute value for the `DRIVER` keyword was longer than 255 characters. |
| IM012   | DRIVER keyword syntax error | (DM) The keyword-value pair for the `DRIVER` keyword contained a syntax error.  
  (DM) The string in *ConnectionString* contained a `FILEDSN` keyword, but the .dsn file did not contain a `DRIVER` keyword or a `DSN` keyword. |
| IM014   | The specified DSN contains an architecture mismatch between the Driver and Application | (DM) 32-bit application uses a DSN connecting to a 64-bit driver; or vice versa. |
| IM015   | Driver's SQLDriverConnect on SQL_HANDLE_DBC_INFO_HANDLE failed | If a driver returns SQL_ERROR, the Driver Manager will return SQL_ERROR to the application and the connection will fail.  
  For more information about SQL_HANDLE_DBC_INFO_TOKEN, see *Developing Connection-Pool Awareness in an ODBC Driver*. |
| IM017   | Polling is disabled in asynchronous notification mode | Whenever the notification model is used, polling is disabled. |
| IM018   | SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle. | If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation. |
| S1118   | Driver does not support asynchronous notification | When the driver does not support asynchronous notification, you cannot set SQL_ATTR_ASYNC_DBC_EVENT or SQL_ATTR_ASYNC_DBC_RETCODE_PTR. |
A connection string has the following syntax:

```
connection-string ::= empty-string[;] | attribute[;] | attribute; connection-string
empty-string ::= attribute ::= attribute-keyword=attribute-value | DRIVER=\{attribute-value\}\}
attribute-keyword ::= DSN | UID | PWD | driver-defined-attribute-keyword
attribute-value ::= character-string
driver-defined-attribute-keyword ::= identifier
```

where character-string has zero or more characters; identifier has one or more characters; attribute-keyword is not case-sensitive; attribute-value may be case-sensitive; and the value of the DSN keyword does not consist solely of blanks.

Because of connection string and initialization file grammar, keywords and attribute values that contain the characters \[\]{}(),;?*!=@ not enclosed with braces should be avoided. The value of the DSN keyword cannot consist only of blanks and should not contain leading blanks. Because of the grammar of the system information, keywords and data source names cannot contain the backslash (\) character.

Applications do not have to add braces around the attribute value after the DRIVER keyword unless the attribute contains a semicolon (;), in which case the braces are required. If the attribute value that the driver receives includes braces, the driver should not remove them but they should be part of the returned connection string.

A DSN or connection string value enclosed with braces (\{}\}) containing any of the characters \[\]{}(),;?*!=@ is passed intact to the driver. However, when using these characters in a keyword, the Driver Manager returns an error when working with file DSNs but passes the connection string to the driver for regular connection strings. Avoid using embedded braces in a keyword value.

The connection string may include any number of driver-defined keywords. Because the DRIVER keyword does not use information from the system information, the driver must define enough keywords so that a driver can connect to a data source using only the information in the connection string. (For more information, see "Driver Guidelines," later in this section.) The driver defines which keywords are required to connect to the data source.

The following table describes the attribute values of the DSN, FILEDSN, DRIVER, UID, PWD, and SAVEFILE keywords.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Attribute value description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DSN</strong></td>
<td>Name of a data source as returned by SQLDataSources or the data sources dialog box of SQLDriverConnect.</td>
</tr>
<tr>
<td><strong>FILEDSN</strong></td>
<td>Name of a .dsn file from which a connection string will be built for the data source. These data sources are called file data sources.</td>
</tr>
<tr>
<td><strong>DRIVER</strong></td>
<td>Description of the driver as returned by the SQLDrivers function. For example, Rdb or SQL Server.</td>
</tr>
</tbody>
</table>
For information about how an application chooses a data source or driver, see Choosing a Data Source or Driver.

If any keywords are repeated in the connection string, the driver uses the value associated with the first occurrence of the keyword. If the DSN and DRIVER keywords are included in the same connection string, the Driver Manager and the driver use whichever keyword appears first.

The FILEDSN and DSN keywords are mutually exclusive: whichever keyword appears first is used, and the one that appears second is ignored. The FILEDSN and DRIVER keywords, on the other hand, are not mutually exclusive. If any keyword appears in a connection string with FILEDSN, then the attribute value of the keyword in the connection string is used rather than the attribute value of the same keyword in the .dsn file.

If the FILEDSN keyword is used, the keywords specified in a .dsn file are used to create a connection string. (For more information, see "File Data Sources," later in this section.) The UID keyword is optional; a .dsn file may be created with only the DRIVER keyword. The PWD keyword is not stored in a .dsn file. The default directory for saving and loading a .dsn file will be a combination of the path specified by CommonFileDir in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion and "ODBC\DataSources". (If CommonFileDir were "C:\Program Files\Common Files", the default directory would be "C:\Program Files\Common Files\ODBC\Data Sources".)

A .dsn file can be manipulated directly by calling the SQLReadFileDSN and SQLWriteFileDSN functions in the installer DLL.

If the SAVEFILE keyword is used, the attribute values of keywords used in making the present, successful connection will be saved as a .dsn file with the name of the attribute value of the SAVEFILE keyword. The SAVEFILE keyword must be used in conjunction with the DRIVER keyword, the FILEDSN keyword, or both, or the function returns SQL_SUCCESS_WITH_INFO with SQLSTATE 01S09 (Invalid keyword). The SAVEFILE keyword must appear before the DRIVER keyword in the connection string, or the results will be undefined.

Driver Manager Guidelines

The Driver Manager constructs a connection string to pass to the driver in the InConnectionString argument of the driver’s SQLDriverConnect function. The Driver Manager does not modify the InConnectionString argument passed to it by the application.

The action of the Driver Manager is based on the value of the DriverCompletion argument:

- SQL_DRIVER_PROMPT: If the connection string does not contain either the DRIVER, DSN, or FILEDSN keyword, the Driver Manager displays the Data Sources dialog box. It constructs a
connection string from the data source name returned by the dialog box and any other keywords passed to it by the application. If the data source name returned by the dialog box is empty, the Driver Manager specifies the keyword-value pair DSN=Default. (This dialog box will not display a data source with the name "Default".)

- SQL_DRIVER_COMPLETE or SQL_DRIVER_COMPLETE_REQUIRED: If the connection string specified by the application includes the DSN keyword, the Driver Manager copies the connection string specified by the application. Otherwise, it takes the same actions as it does when DriverCompletion is SQL_DRIVER_PROMPT.

- SQL_DRIVER_NOPROMPT: The Driver Manager copies the connection string specified by the application.

If the connection string specified by the application contains the DRIVER keyword, the Driver Manager copies the connection string specified by the application.

Using the connection string it has constructed, the Driver Manager determines which driver to use, connects to that driver, and passes the connection string it has constructed to the driver; for more information about the interaction of the Driver Manager and the driver, see the "Comments" section in SQLConnect Function. If the connection string does not contain the DRIVER keyword, the Driver Manager determines which driver to use as follows:

1. If the connection string contains the DSN keyword, the Driver Manager retrieves the driver associated with the data source from the system information.

2. If the connection string does not contain the DSN keyword or the data source is not found, the Driver Manager retrieves the driver associated with the Default data source from the system information. (For more information, see Default Subkey.) The Driver Manager changes the value of the DSN keyword in the connection string to "DEFAULT".

3. If the DSN keyword in the connection string is set to "DEFAULT", the Driver Manager retrieves the driver associated with the Default data source from the system information.

4. If the data source is not found and the Default data source is not found, the Driver Manager returns SQL_ERROR with SQLSTATE IM002 (Data source not found and no default driver specified).

**File Data Sources**

If the connection string specified by the application in the call to SQLDriverConnect contains the FILEDSN keyword, and this keyword is not superseded by either the DSN or DRIVER keyword, then the Driver Manager creates a connection string using the information in the .dsn file and the InConnectionString argument. The Driver Manager proceeds as follows:

1. Checks whether the file name of the .dsn file is valid. If not, it returns SQL_ERROR with SQLSTATE IM014 (Invalid name of file DSN). If the file name is an empty string ("") and SQL_DRIVER_NOPROMPT is not specified, then the File-Open dialog box is displayed. If the file name contains a valid path but no file name or an invalid file name, and SQL_DRIVER_NOPROMPT is not specified, then the File-Open dialog box is displayed with the current directory set to the one specified in the file name. If the file name is an empty string ("") or the file name contains a valid path but no file name or an invalid file name, and SQL_DRIVER_NOPROMPT is specified, then SQL_ERROR is returned with SQLSTATE IM014.
2. Reads all keywords in the [ODBC] section of the .dsn file. If the DRIVER keyword is not present, it returns SQL_ERROR with SQLSTATE IM012 (Driver keyword syntax error), except where the .dsn file is unshareable and thus contains only the DSN keyword.

If the file data source is unshareable, the Driver Manager reads the value of the DSN keyword and connects as necessary to the user or system data source pointed to by the unshareable file data source. Steps 3 through 5 are not performed.

3. Constructs a connection string for the driver. The driver connection string is the union of the keywords specified in the .dsn file and those specified in the original application connection string. Rules for the construction of the driver connection string where keywords overlap are as follows:

   - If the DRIVER keyword exists in the application connection string and the drivers specified by the DRIVER keywords are not the same in the .dsn file and the application connection string, then the driver information in the .dsn file is ignored and the driver information in the application connection string is used. If the drivers specified by the DRIVER keyword are the same in the .dsn file and the application's connection string, then where all keywords overlap, those specified in the application connection string have precedence over those specified in the .dsn file.

   - In the new connection string, the FILEDSN keyword is eliminated.

4. Loads the driver by looking in the registry entry HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\<Driver Name>\Driver where <Driver Name> is specified by the DRIVER keyword.

5. Passes the driver the new connections string.

For examples of .dsn files, see Connecting Using File Data Sources.

SAVEFILE Keyword

If the connection string specified by the application contains the SAVEFILE keyword, then the Driver Manager saves the connection string in a .dsn file. The Driver Manager proceeds as follows:

1. Checks whether the file name of the .dsn file included as the attribute value of the SAVEFILE keyword is valid. If not, it returns SQL_ERROR with SQLSTATE IM014 (Invalid name of file DSN). The validity of the file name is determined by standard system naming rules. If the file name is an empty string (""") and the DriverCompletion argument is not SQL_DRIVER_NOPROMPT, then the file name is valid. If the file name already exists, then if DriverCompletion is SQL_DRIVER_NOPROMPT, the file is overwritten. If DriverCompletion is SQL_DRIVER_PROMPT, SQL_DRIVER_COMPLETE, or SQL_DRIVER_COMPLETE_REQUIRED, a dialog box prompts the user to specify whether the file should be overwritten. If No is entered, then the File-Save dialog box appears.

2. If the driver returns SQL_SUCCESS and the file name was not an empty string, then the Driver Manager writes the connection information returned in the OutConnectionString argument to the specified file with the format specified in the "Connection Strings" section earlier in this section.

3. If the driver returns SQL_SUCCESS and the file name was an empty string (""), then the Driver Manager calls the File-Save common dialog box with the hwnd specified and writes the file.
connection information returned in `OutConnectionString` to the file specified in the File-Save common dialog box with the format specified in the "Connection Strings" section earlier in this section.

4. If the driver returns SQL_SUCCESS, it returns the `OutConnectionString` argument containing the connection string to the application.

5. If the driver returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, then the Driver Manager returns the SQLSTATE to the application.

Driver Guidelines

The driver checks whether the connection string passed to it by the Driver Manager contains the `DSN` or `DRIVER` keyword. If the connection string contains the `DRIVER` keyword, the driver cannot retrieve information about the data source from the system information. If the connection string contains the `DSN` keyword or does not contain either the `DSN` or the `DRIVER` keyword, the driver can retrieve information about the data source from the system information as follows:

1. If the connection string contains the `DSN` keyword, the driver retrieves the information for the specified data source.

2. If the connection string does not contain the `DSN` keyword, the specified data source is not found, or the `DSN` keyword is set to "DEFAULT", the driver retrieves the information for the Default data source.

The driver uses any information it retrieves from the system information to augment the information passed to it in the connection string. If the information in the system information duplicates information in the connection string, the driver uses the information in the connection string.

Based on the value of `DriverCompletion`, the driver prompts the user for connection information, such as the user ID and password, and connects to the data source:

- **SQL_DRIVER_PROMPT:** The driver displays a dialog box, using the values from the connection string and system information (if any) as initial values. When the user exits the dialog box, the driver connects to the data source. It also constructs a connection string from the value of the `DSN` or `DRIVER` keyword in `*InConnectionString` and the information returned from the dialog box. It places this connection string in the `*OutConnectionString` buffer.

- **SQL_DRIVER_COMPLETE** or **SQL_DRIVER_COMPLETE_REQUIRED:** If the connection string contains enough information, and that information is correct, the driver connects to the data source and copies `*InConnectionString` to `*OutConnectionString`. If any information is missing or incorrect, the driver takes the same actions as it does when `DriverCompletion` is SQL_DRIVER_PROMPT, except that if `DriverCompletion` is SQL_DRIVER_COMPLETE_REQUIRED, the driver disables the controls for any information not required to connect to the data source.

- **SQL_DRIVER_NOPROMPT:** If the connection string contains enough information, the driver connects to the data source and copies `*InConnectionString` to `*OutConnectionString`. Otherwise, the driver returns SQL_ERROR for `SQLDriverConnect`.

On successful connection to the data source, the driver also sets `*StringLength2Ptr` to the length of the output connection string that is available to return in `*OutConnectionString`. 


If the user cancels a dialog box presented by the Driver Manager or the driver, **SQLDriverConnect** returns SQL_NO_DATA.

For information about how the Driver Manager and the driver interact during the connection process, see [SQLConnect Function](#).

If a driver supports **SQLDriverConnect**, the driver keyword section of the system information for the driver must contain the **ConnectFunctions** keyword with the second character set to "Y".

### Connecting When Connection Pooling Is Enabled

Connection pooling allows an application to reuse a connection that has already been created. When **SQLDriverConnect** is called, the Driver Manager attempts to make the connection using a connection that is part of a pool of connections in an environment that has been designated for connection pooling. For more information on connection pooling, see [SQLConnect Function](#).

An application can set SQL_ATTR_RESET_CONNECTION before calling SQLDisconnect on a connection where pooling is enabled. For more information, see [SQLSetConnectAttr Function](#).

The following restrictions apply when an application calls **SQLDriverConnect** to connect to a pooled connection:

- No connection pooling processing is performed when the **SAVEFILE** keyword is specified in the connection string.

- If connection pooling is enabled, **SQLDriverConnect** can be called only with a DriverCompletion argument of SQL_DRIVER_NOPROMPT; if **SQLDriverConnect** is called with any other DriverCompletion, SQLSTATE HY110 (Invalid driver completion) will be returned.

### Connection Attributes

The SQL_ATTR_LOGIN_TIMEOUT connection attribute, set using **SQLSetConnectAttr**, defines the number of seconds to wait for a login request to complete with a successful connection by the driver before returning to the application. If the user is prompted to complete the connection string, a waiting period for each login request begins when the driver starts the connection process.

The driver opens the connection in SQL_MODE_READ_WRITE access mode by default. To set the access mode to SQL_MODE_READ_ONLY, the application must call **SQLSetConnectAttr** with the SQL_ATTR_ACCESS_MODE attribute prior to calling **SQLDriverConnect**.

If a default translation library is specified in the system information for the data source, the driver loads it. A different translation library can be loaded by calling **SQLSetConnectAttr** with the SQL_ATTR_TRANSLATE_LIB attribute. A translation option can be specified by calling **SQLSetConnectAttr** with the SQL_ATTR_TRANSLATE_OPTION option.

For more information, see [Connecting with SQLDriverConnect](#).

```c++
#include <windows.h>
```

---

// SQLDriverConnect_ref.cpp
// compile with: odbc32.lib user32.lib
#include <windows.h>
Also see Sample ODBC Program.
## Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocating a handle</td>
<td>SQLAllocHandle Function</td>
</tr>
<tr>
<td>Discovering and enumerating values required to connect to a data source</td>
<td>SQLBrowseConnect Function</td>
</tr>
<tr>
<td>Connecting to a data source</td>
<td>SQLConnect Function</td>
</tr>
<tr>
<td>Disconnecting from a data source</td>
<td>SQLDisconnect Function</td>
</tr>
<tr>
<td>Returning driver descriptions and attributes</td>
<td>SQLDrivers Function</td>
</tr>
<tr>
<td>Freeing a handle</td>
<td>SQLFreeHandle Function</td>
</tr>
<tr>
<td>Setting a connection attribute</td>
<td>SQLSetConnectAttr Function</td>
</tr>
</tbody>
</table>

## See Also

- ODBC API Reference
- ODBC Header Files

### Conformance

**Version Introduced:** ODBC 2.0  
**Standards Compliance:** ODBC

### Summary

**SQLDrivers** lists driver descriptions and driver attribute keywords. This function is implemented only by the Driver Manager.

### Syntax

```c
SQLRETURN SQLDrivers(
    SQLHENV EnvironmentHandle,
    SQLUSMALLINT Direction,
);```
Arguments

EnvironmentHandle
[Input] Environment handle.

Direction
[Input] Determines whether the Driver Manager fetches the next driver description in the list (SQL_FETCH_NEXT) or whether the search starts from the beginning of the list (SQL_FETCH_FIRST).

DriverDescription
[Output] Pointer to a buffer in which to return the driver description.

If DriverDescription is NULL, DescriptionLengthPtr will still return the total number of characters (excluding the null-termination character for character data) available to return in the buffer pointed to by DriverDescription.

BufferLength1
[Input] Length of the *DriverDescription buffer, in characters.

DescriptionLengthPtr
[Output] Pointer to a buffer in which to return the total number of characters (excluding the null-termination character) available to return in *DriverDescription. If the number of characters available to return is greater than or equal to BufferLength1, the driver description in *DriverDescription is truncated to BufferLength1 minus the length of a null-termination character.

DriverAttributes
[Output] Pointer to a buffer in which to return the list of driver attribute value pairs (see "Comments").

If DriverAttributes is NULL, AttributesLengthPtr will still return the total number of bytes (excluding the null-termination character for character data) available to return in the buffer pointed to by DriverAttributes.

BufferLength2
[Input] Length of the *DriverAttributes buffer, in characters. If the *DriverDescription value is a Unicode string (when calling SQLDriversW), the BufferLength argument must be an even number.

AttributesLengthPtr
[Output] Pointer to a buffer in which to return the total number of bytes (excluding the null-termination byte) available to return in *DriverAttributes. If the number of bytes available to return is greater than or equal to BufferLength2, the list of attribute value pairs in *DriverAttributes is truncated to BufferLength2 minus the length of the null-termination character.

| SQLCHAR * | DriverDescription, |
| SQLSMALLINT | BufferLength1, |
| SQLSMALLINT * | DescriptionLengthPtr, |
| SQLCHAR * | DriverAttributes, |
| SQLSMALLINT | BufferLength2, |
| SQLSMALLINT * | AttributesLengthPtr |);
Returns
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLDrivers returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_ENV and a Handle of EnvironmentHandle. The following table lists the SQLSTATE values typically returned by SQLDrivers and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>(DM) Driver Manager–specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>(DM) The buffer *DriverDescription was not large enough to return the complete driver description. Therefore, the description was truncated. The length of the complete driver description is returned in *DescriptionLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The buffer *DriverAttributes was not large enough to return the complete list of attribute value pairs. Therefore, the list was truncated. The length of the untruncated list of attribute value pairs is returned in *AttributesLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>(DM) The Driver Manager was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer</td>
<td>(DM) The value specified for argument BufferLength1 was</td>
</tr>
</tbody>
</table>
Comments

SQLDrivers returns the driver description in the *DriverDescription buffer. It returns additional information about the driver in the *DriverAttributes buffer as a list of keyword-value pairs. All keywords listed in the system information for drivers will be returned for all drivers, except for CreateDSN, which is used to prompt creation of data sources and therefore is optional. Each pair is terminated with a null byte, and the complete list is terminated with a null byte (that is, two null bytes mark the end of the list). For example, a file-based driver using C syntax might return the following list of attributes ("\0" represents a null character):

```
FileUsage=1\0FileExtns=*.dbf\0\0
```

If *DriverAttributes is not large enough to hold the entire list, the list is truncated, SQLDrivers returns SQLSTATE 01004 (Data truncated), and the length of the list (excluding the final null-termination byte) is returned in *AttributesLengthPtr.

Driver attribute keywords are added from the system information when the driver is installed. For more information, see Installing ODBC Components.

An application can call SQLDrivers multiple times to retrieve all driver descriptions. The Driver Manager retrieves this information from the system information. When there are no more driver descriptions, SQLDrivers returns SQL_NO_DATA. If SQLDrivers is called with SQL_FETCH_NEXT immediately after it returns SQL_NO_DATA, it returns the first driver description. For information about how an application uses the information returned by SQLDrivers, see Choosing a Data Source or Driver.

If SQL_FETCH_NEXT is passed to SQLDrivers the very first time it is called, SQLDrivers returns the first data source name.

Because SQLDrivers is implemented in the Driver Manager, it is supported for all drivers regardless of a particular driver’s standards compliance.

Related Functions
For information about | See
--- | ---
Discovering and listing values required to connect to a data source | SQLBrowseConnect Function
Connecting to a data source | SQLConnect Function
Returning data source names | SQLDataSources Function
Connecting to a data source using a connection string or dialog box | SQLDriverConnect Function

See Also
ODBC API Reference
ODBC Header Files

© 2016 Microsoft

**SQLEndTran Function**

**Conformance**
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

**Summary**
`SQLEndTran` requests a commit or rollback operation for all active operations on all statements associated with a connection. `SQLEndTran` can also request that a commit or rollback operation be performed for all connections associated with an environment.

**Note**
For more information about what the Driver Manager maps this function to when an ODBC 3.x application is working with an ODBC 2.x driver, see [Mapping Replacement Functions for Backward Compatibility of Applications](#).

**Syntax**

```c
SQLRETURN SQLEndTran(
    SQLSMALLINT HandleType,
    SQLHANDLE Handle,
    SQLSMALLINT CompletionType);
```
Arguments

(HandleType [Input] Handle type identifier. Contains either SQL_HANDLE_ENV (if Handle is an environment handle) or SQL_HANDLE_DBC (if Handle is a connection handle).

Handle [Input] The handle, of the type indicated by HandleType, indicating the scope of the transaction. See "Comments" for more information.

CompletionType [Input] One of the following two values:

SQL_COMMIT SQL_ROLLBACK

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_STILL_EXECUTING.

Diagnostics

When SQLEndTran returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with the appropriate HandleType and Handle. The following table lists the SQLSTATE values commonly returned by SQLEndTran and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
<td>(DM) The HandleType was SQL_HANDLE_DBC, and the Handle was not in a connected state.</td>
</tr>
<tr>
<td>08007</td>
<td>Connection failure during transaction</td>
<td>The HandleType was SQL_HANDLE_DBC, and the connection associated with the Handle failed during the execution of the function, and it cannot be determined whether the requested COMMIT or ROLLBACK occurred before the failure.</td>
</tr>
<tr>
<td>25S01</td>
<td>Transaction state unknown</td>
<td>One or more of the connections in Handle failed to complete the transaction with the outcome specified, and the outcome is unknown.</td>
</tr>
<tr>
<td>25S02</td>
<td>Transaction is still active</td>
<td>The driver was not able to guarantee that all work in the transaction completed successfully.</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Error Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>25S03</td>
<td>Transaction is rolled back</td>
<td>The driver was not able to guarantee that all work in the global transaction could be completed atomically, and all work in the transaction active in <em>Handle</em> was rolled back.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back due to a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40002</td>
<td>Integrity constraint violation</td>
<td>The <code>CompletionType</code> was <code>SQL_COMMIT</code>, and the commitment of changes caused integrity constraint violation. As a result, the transaction was rolled back.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <code>*szMessageText</code> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <em>ConnectionHandle</em>. The function was called, and before it finished executing <code>SQLCancelHandle</code> function was called on the <em>ConnectionHandle</em>. Then the function was called again on the <em>ConnectionHandle</em>. The function was called, and before it finished executing <code>SQLCancelHandle</code> was called on the <em>ConnectionHandle</em> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for a statement handle associated with the <em>ConnectionHandle</em> and was still executing when <code>SQLEndTran</code> was called. (DM) An asynchronously executing function (not this one) was called for the <em>ConnectionHandle</em> and was still executing when this function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for a statement handle associated with the <em>ConnectionHandle</em> and returned <code>SQL_NEED_DATA</code>. This function was called before data was sent for all data-at-execution parameters or columns. (DM) An asynchronously executing function (not this one) was called for the <em>Handle</em> with <em>HandleType</em> set to <code>SQL_HANDLE_DBC</code> and was still executing when this function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for one of the statement handles associated with <em>Handle</em> and returned <code>SQL_PARAM_DATA_AVAILABLE</code>.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY012</td>
<td>Invalid transaction operation code</td>
<td>(DM) The value specified for the argument CompletionType was neither SQL_COMMIT nor SQL_ROLLBACK.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute/option identifier</td>
<td>(DM) The value specified for the argument HandleType was neither SQL_HANDLE_ENV nor SQL_HANDLE_DBC.</td>
</tr>
<tr>
<td>HY115</td>
<td>SQLEndTran is not allowed for an environment that contains a connection with asynchronous function execution enabled</td>
<td>(DM) HandleType cannot be set to SQL_HANDLE_ENV if asynchronous execution of connection functions is enabled for a connection in the environment.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, refer to the Comments section of this topic.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The driver or data source does not support the ROLLBACK operation.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the ConnectionHandle does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><strong>SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.</strong></td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

**Comments**
For an ODBC 3.x driver, if HandleType is SQL_HANDLE_ENV and Handle is a valid environment handle, then the Driver Manager will call SQLEndTran in each driver associated with the environment. The Handle argument for the call to a driver will be the driver’s environment handle. For an ODBC 2.x driver, if HandleType is SQL_HANDLE_ENV and Handle is a valid environment handle, and there are multiple connections in a connected state in that environment, then the Driver Manager will call SQLTransact in the driver once for each connection in a connected state in that environment. The Handle argument in each call will be the connection’s handle. In either case, the driver will attempt to commit or roll back transactions, depending on the value of CompletionType, on all connections that are in a connected state on that environment. Connections that are not active do not affect the transaction.

**Note**

SQLEndTran cannot be used to commit or roll back transactions on a shared environment. SQLSTATE HY092 (Invalid attribute/option identifier) will be returned if SQLEndTran is called with Handle set to either the handle of a shared environment or the handle of a connection on a shared environment.

The Driver Manager will return SQL_SUCCESS only if it receives SQL_SUCCESS for each connection. If the Driver Manager receives SQL_ERROR on one or more connections, it returns SQL_ERROR to the application, and the diagnostic information is placed in the diagnostic data structure of the environment. To determine which connection or connections failed during the commit or rollback operation, the application can call SQLGetDiagRec for each connection.

**Note**

The Driver Manager does not simulate a global transaction across all connections and therefore does not use two-phase commit protocols.

If CompletionType is SQL_COMMIT, SQLEndTran issues a commit request for all active operations on any statement associated with an affected connection. If CompletionType is SQL_ROLLBACK, SQLEndTran issues a rollback request for all active operations on any statement associated with an affected connection. If no transactions are active, SQLEndTran returns SQL_SUCCESS with no effect on any data sources. For more information, see Committing and Rolling Back Transactions.

If the driver is in manual-commit mode (by calling SQLSetConnectAttr with the SQL_ATTR_AUTOCOMMIT attribute set to SQL_AUTOCOMMIT_OFF), a new transaction is implicitly started when an SQL statement that can be contained within a transaction is executed against the current data source. For more information, see Commit Mode.

To determine how transaction operations affect cursors, an application calls SQLGetInfo with the SQL_CURSOR_ROLLBACK_BEHAVIOR and SQL_CURSOR_COMMIT_BEHAVIOR options. For more information, see the following paragraphs and also see Effect of Transactions on Cursors and Prepared Statements.

If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value equals SQL_CB_DELETE, SQLEndTran closes and deletes all open cursors on all statements associated with the connection and discards all pending results. SQLEndTran leaves any statement present in an allocated (unprepared) state; the application can reuse them for subsequent SQL requests or can call SQLFreeStmt or SQLFreeHandle with a HandleType of SQL_HANDLE_STMT to deallocate them.

If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value equals SQL_CB_CLOSE, SQLEndTran closes all open cursors on all statements associated with the connection. SQLEndTran leaves any statement present in a prepared state; the application can call SQLExecute for a statement associated with the connection without first calling SQLPrepare.
If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value equals SQL_CB_PRESERVE, SQLEndTran does not affect open cursors associated with the connection. Cursors remain at the row they pointed to prior to the call to SQLEndTran.

For drivers and data sources that support transactions, calling SQLEndTran with either SQL_COMMIT or SQL_ROLLBACK when no transaction is active returns SQL_SUCCESS (indicating that there is no work to be committed or rolled back) and has no effect on the data source.

When a driver is in autocommit mode, the Driver Manager does not call SQLEndTran in the driver. SQLEndTran always returns SQL_SUCCESS regardless of whether it is called with a CompletionType of SQL_COMMIT or SQL_ROLLBACK.

Drivers or data sources that do not support transactions (SQLGetInfo option SQL_TXN_CAPABLE is SQL_TC_NONE) are effectively always in autocommit mode and therefore always return SQL_SUCCESS for SQLEndTran whether or not they are called with a CompletionType of SQL_COMMIT or SQL_ROLLBACK. Such drivers and data sources do not actually roll back transactions when requested to do so.

**Suspended State**

In Driver Managers that were released before Windows 7, a transaction was active if SQLEndTran returned SQL_ERROR from the driver. However, it was possible that the transaction had been successfully committed on the server, but the driver on the client had not been notified (for example, because a network error occurred). This would leave the connection in a bad state. Starting with Windows 7, when SQLEndTran returns SQL_ERROR, the connection might be in a suspended state. In a suspended state, it is possible to call read-only functions. Eventually, the application should call SQLDisconnect on a suspended connection to release resources.

If all of the following conditions are true, the connection will be put into a suspended state:

- The driver returns SQL_ERROR from SQLEndTran.
- The driver is ODBC version 3.8, or later.
- The application version is 3.8 or later; or the recompiled ODBC 2.x or 3.x application successfully cancels the SQLEndTran function through SQLCancelHandle.
- The driver did not return one of the following messages, which confirm that the transaction did not complete:
  - 25S03: Transaction is rolled back
  - 40001: Serialization failure
  - 40002: Integrity constraint
  - HYC00: Optional feature not implemented

If SQLEndTran was called on an environment handle and one of its connections met the above conditions, all connections connecting to the same driver will be put into the suspended state.

After an application calls SQLDisconnect on a suspended connection, the connection can be used to reconnect to another data source or the same data source.
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canceling a function running asynchronously on a connection handle.</td>
<td>SQLCancelHandle Function</td>
</tr>
<tr>
<td>Returning information about a driver or data source</td>
<td>SQLGetInfo Function</td>
</tr>
<tr>
<td>Freeing a handle</td>
<td>SQLFreeHandle Function</td>
</tr>
<tr>
<td>Freeing a statement handle</td>
<td>SQLFreeStmt Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files
Asynchronous Execution (Polling Method)

© 2016 Microsoft

SQLError Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

Summary
SQLError returns error or status information.

For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLExecDirect Function
Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLExecDirect executes a preparable statement, using the current values of the parameter marker variables if any parameters exist in the statement. SQLExecDirect is the fastest way to submit an SQL statement for one-time execution.

Syntax

```c
SQLRETURN SQLExecDirect(
    SQLHSTMT StatementHandle,
    SQLCHAR * StatementText,
    SQLINTEGER TextLength);
```

Arguments

StatementHandle
    [Input] Statement handle.

StatementText
    [Input] SQL statement to be executed.

TextLength
    [Input] Length of *StatementText in characters.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, SQL_NO_DATA, SQL_INVALID_HANDLE, or SQL_PARAM_DATA_AVAILABLE.

Diagnostics

When SQLExecDirect returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLExecDirect and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>01001</td>
<td>Cursor operation conflict</td>
<td>*StatementText contained a positioned update or delete statement, and no rows or more than one row were updated or deleted. (For more information about updates to more than one row, see the description of the SQL_ATTR_SIMULATE_CURSOR Attribute in SQLSetStmtAttr.) (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01003</td>
<td>NULL value eliminated in set function</td>
<td>The argument StatementText contained a set function (such as AVG, MAX, MIN, and so on), but not the COUNT set function, and NULL argument values were eliminated before the function was applied. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>String or binary data returned for an input/output or output parameter resulted in the truncation of nonblank character or non-NULL binary data. If it was a string value, it was right-truncated. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01006</td>
<td>Privilege not revoked</td>
<td>*StatementText contained a REVOKE statement, and the user did not have the specified privilege. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01007</td>
<td>Privilege not granted</td>
<td>*StatementText was a GRANT statement, and the user could not be granted the specified privilege.</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed</td>
<td>A specified statement attribute was invalid because of implementation working conditions, so a similar value was temporarily substituted. (SQLGetStmtAttr can be called to determine what the temporarily substituted value is.) The substitute value is valid for the StatementHandle until the cursor is closed, at which point the statement attribute reverts to its previous value. The statement attributes that can be changed are: SQL_ATTR_CONCURRENCY SQL_ATTR_CURSOR_TYPE SQL_ATTR_KEYSET_SIZE SQL_ATTR_MAX_LENGTH SQL_ATTR_MAX_ROWS SQL_ATTR_QUERY_TIMEOUT SQL_ATTR_SIMULATE_CURSOR (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S07</td>
<td>Fractional truncation</td>
<td>The data returned for an input/output or output parameter was truncated such that the fractional part of a numeric data type was truncated or the fractional portion of the time component of a time, timestamp, or interval data type was truncated. (Function returns SQL_SUCCESS_WITHINFO.)</td>
</tr>
<tr>
<td>07002</td>
<td>COUNT field incorrect</td>
<td>The number of parameters specified in SQLBindParameter was less than the number of parameters in the SQL statement contained in *StatementText.</td>
</tr>
</tbody>
</table>
**SQLBindParameter** was called with `ParameterValuePtr` set to a null pointer, `StrLen_or_IndPtr` not set to `SQL_NULL_DATA` or `SQL_DATA_AT_EXEC`, and `InputOutputType` not set to `SQL_PARAM_OUTPUT`, so that the number of parameters specified in **SQLBindParameter** was greater than the number of parameters in the SQL statement contained in *StatementText*.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>The data value identified by the <code>ValueType</code> argument in <strong>SQLBindParameter</strong> for the bound parameter could not be converted to the data type identified by the <code>ParameterType</code> argument in <strong>SQLBindParameter</strong>. The data value returned for a parameter bound as <code>SQL_PARAM_INPUT_OUTPUT</code> or <code>SQL_PARAM_OUTPUT</code> could not be converted to the data type identified by the <code>ValueType</code> argument in <strong>SQLBindParameter</strong>. (If the data values for one or more rows could not be converted but one or more rows were successfully returned, this function returns <code>SQL_SUCCESS_WITH_INFO</code>.)</td>
</tr>
<tr>
<td>07007</td>
<td>Restricted parameter value violation</td>
<td>The parameter type <code>SQL_PARAM_INPUT_OUTPUT_STREAM</code> is only used for a parameter that sends and receives data in parts. An input bound buffer is not allowed for this parameter type. This error will occur when the parameter type is <code>SQL_PARAM_INPUT_OUTPUT</code>, and when the <code>*StrLen_or_IndPtr</code> specified in <strong>SQLBindParameter</strong> is not equal to <code>SQL_NULL_DATA</code>, <code>SQL_DEFAULT_PARAM</code>, <code>SQL_LEN_DATA_AT_EXEC(len)</code>, or <code>SQL_DATA_AT_EXEC</code>.</td>
</tr>
<tr>
<td>07S01</td>
<td>Invalid use of default parameter</td>
<td>A parameter value, set with <strong>SQLBindParameter</strong>, was <code>SQL_DEFAULT_PARAM</code>, and the corresponding parameter did not have a default value.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>21S01</td>
<td>Insert value list does not match column list</td>
<td><em>StatementText</em> contained an <strong>INSERT</strong> statement, and the number of values to be inserted did not match the degree of the derived table.</td>
</tr>
<tr>
<td>21S02</td>
<td>Degree of derived table does not match column list</td>
<td><em>StatementText</em> contained a <strong>CREATE VIEW</strong> statement, and the unqualified column list (the number of columns specified for the view in the <code>column-identifier</code> arguments of the SQL statement) contained more names than the number of columns in the derived table defined by the <code>query-specification</code> argument of the SQL statement.</td>
</tr>
<tr>
<td>22001</td>
<td>String data, right truncation</td>
<td>The assignment of a character or binary value to a column resulted in the truncation of nonblank character data or non-null binary data.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>22002</td>
<td>Indicator variable required but not supplied</td>
<td>NULL data was bound to an output parameter whose StrLen_or_IndPtr set by SQLBindParameter was a null pointer.</td>
</tr>
<tr>
<td>22003</td>
<td>Numeric value out of range</td>
<td>*StatementText contained an SQL statement that contained a bound numeric parameter or literal, and the value caused the whole (as opposed to fractional) part of the number to be truncated when assigned to the associated table column. Returning a numeric value (as numeric or string) for one or more input/output or output parameters would have caused the whole (as opposed to fractional) part of the number to be truncated.</td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format</td>
<td>*StatementText contained an SQL statement that contained a date, time, or timestamp structure as a bound parameter, and the parameter was, respectively, an invalid date, time, or timestamp. An input/output or output parameter was bound to a date, time, or timestamp C structure, and a value in the returned parameter was, respectively, an invalid date, time, or timestamp. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>22008</td>
<td>Datetime field overflow</td>
<td>*StatementText contained an SQL statement that contained a datetime expression that, when computed, resulted in a date, time, or timestamp structure that was invalid. A datetime expression computed for an input/output or output parameter resulted in a date, time, or timestamp C structure that was invalid.</td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
<td>*StatementText contained an SQL statement that contained an arithmetic expression that caused division by zero. An arithmetic expression calculated for an input/output or output parameter resulted in division by zero.</td>
</tr>
<tr>
<td>22015</td>
<td>Interval field overflow</td>
<td>*StatementText contained an exact numeric or interval parameter that, when converted to an interval SQL data type, caused a loss of significant digits. *StatementText contained an interval parameter with more than one field that, when converted to a numeric data type in a column, had no representation in the numeric data type. *StatementText contained parameter data that was assigned to an interval SQL type, and there was no representation of the value of the C type in the interval SQL type. Assigning an input/output or output parameter that was an exact numeric or interval SQL type to an interval C type caused a loss of significant digits. When an input/output or output parameter was assigned to</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast</td>
<td><em>StatementText</em> contained a C type that was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type.</td>
</tr>
<tr>
<td></td>
<td>specification</td>
<td>When an input/output or output parameter was returned, the SQL type was an exact or approximate numeric, a datetime, or an interval data type; the C type was SQL_C_CHAR; and the value in the column was not a valid literal of the bound SQL type.</td>
</tr>
<tr>
<td>22019</td>
<td>Invalid escape character</td>
<td><em>StatementText</em> contained an SQL statement that contained a <code>LIKE</code> predicate with an <code>ESCAPE</code> in the <code>WHERE</code> clause, and the length of the escape character following <code>ESCAPE</code> was not equal to 1.</td>
</tr>
<tr>
<td>22025</td>
<td>Invalid escape sequence</td>
<td><em>StatementText</em> contained an SQL statement that contained &quot;<code>LIKE</code> pattern value <code>ESCAPE</code> escape character&quot; in the <code>WHERE</code> clause, and the character following the escape character in the pattern value was not one of &quot;%&quot; or &quot;_&quot;.</td>
</tr>
<tr>
<td>23000</td>
<td>Integrity constraint violation</td>
<td><em>StatementText</em> contained an SQL statement that contained a parameter or literal. The parameter value was NULL for a column defined as NOT NULL in the associated table column, a duplicate value was supplied for a column constrained to contain only unique values, or some other integrity constraint was violated.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>A cursor was positioned on the <code>StatementHandle</code> by <code>SQLFetch</code> or <code>SQLFetchScroll</code>. This error is returned by the Driver Manager if <code>SQLFetch</code> or <code>SQLFetchScroll</code> has not returned SQL_NO_DATA, and is returned by the driver if <code>SQLFetch</code> or <code>SQLFetchScroll</code> has returned SQL_NO_DATA.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A cursor was open but not positioned on the <code>StatementHandle</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>StatementText</em> contained a positioned update or delete statement, and the cursor was positioned before the start of the result set or after the end of the result set.</td>
</tr>
<tr>
<td>34000</td>
<td>Invalid cursor name</td>
<td><em>StatementText</em> contained a positioned update or delete statement, and the cursor referenced by the statement being executed was not open.</td>
</tr>
<tr>
<td>3D000</td>
<td>Invalid catalog name</td>
<td>The catalog name specified in <code>StatementText</code> was invalid.</td>
</tr>
<tr>
<td>3F000</td>
<td>Invalid schema name</td>
<td>The schema name specified in <code>StatementText</code> was invalid.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back due to a resource deadlock.</td>
</tr>
</tbody>
</table>
with another transaction.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>42000</td>
<td>Syntax error or access violation</td>
<td><em>StatementText</em> contained an SQL statement that was not preparable or contained a syntax error. The user did not have permission to execute the SQL statement contained in <em>StatementText</em>.</td>
</tr>
<tr>
<td>42S01</td>
<td>Base table or view already exists</td>
<td><em>StatementText</em> contained a <strong>CREATE TABLE</strong> or <strong>CREATE VIEW</strong> statement, and the table name or view name specified already exists.</td>
</tr>
</tbody>
</table>
| 42S02      | Base table or view not found | *StatementText* contained a **DROP TABLE** or a **DROP VIEW** statement, and the specified table name or view name did not exist.  
*StatementText* contained an **ALTER TABLE** statement, and the specified table name did not exist.  
*StatementText* contained a **CREATE VIEW** statement, and a table name or view name defined by the query specification did not exist.  
*StatementText* contained a **CREATE INDEX** statement, and the specified table name did not exist.  
*StatementText* contained a **GRANT** or **REVOKE** statement, and the specified table name or view name did not exist.  
*StatementText* contained a **SELECT** statement, and a specified table name or view name did not exist.  
*StatementText* contained a **DELETE, INSERT, or UPDATE** statement, and the specified table name did not exist.  
*StatementText* contained a **CREATE TABLE** statement, and a table specified in a constraint (referencing a table other than the one being created) did not exist.  
*StatementText* contained a **CREATE SCHEMA** statement, and a specified table name or view name did not exist. |
| 42S11      | Index already exists | *StatementText* contained a **CREATE INDEX** statement, and the specified index name already existed.  
*StatementText* contained a **CREATE SCHEMA** statement, and the specified index name already existed. |
<p>| 42S12      | Index not found | <em>StatementText</em> contained a <strong>DROP INDEX</strong> statement, and the specified index name did not exist. |
| 42S21      | Column already | <em>StatementText</em> contained an <strong>ALTER TABLE</strong> statement, and |</p>
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists</td>
<td>the column specified in the <strong>ADD</strong> clause is not unique or identifies an existing column in the base table.</td>
<td></td>
</tr>
<tr>
<td>42S22</td>
<td>Column not found</td>
<td><em>StatementText</em> contained a <strong>CREATE INDEX</strong> statement, and one or more of the column names specified in the column list did not exist.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>StatementText</em> contained a <strong>GRANT</strong> or <strong>REVOKE</strong> statement, and a specified column name did not exist.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>StatementText</em> contained a <strong>SELECT</strong>, <strong>DELETE</strong>, <strong>INSERT</strong>, or <strong>UPDATE</strong> statement, and a specified column name did not exist.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>StatementText</em> contained a <strong>CREATE TABLE</strong> statement, and a column specified in a constraint (referencing a table other than the one being created) did not exist.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>StatementText</em> contained a <strong>CREATE SCHEMA</strong> statement, and a specified column name did not exist.</td>
</tr>
<tr>
<td>44000</td>
<td>WITH CHECK OPTION violation</td>
<td>The argument <em>StatementText</em> contained an <strong>INSERT</strong> statement performed on a viewed table or a table derived from the viewed table that was created by specifying <strong>WITH CHECK OPTION</strong>, such that one or more rows affected by the <strong>INSERT</strong> statement will no longer be present in the viewed table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The argument <em>StatementText</em> contained an <strong>UPDATE</strong> statement performed on a viewed table or a table derived from the viewed table that was created by specifying <strong>WITH CHECK OPTION</strong>, such that one or more rows affected by the <strong>UPDATE</strong> statement will no longer be present in the viewed table.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific <strong>SQLSTATE</strong> and for which no implementation-specific <strong>SQLSTATE</strong> was defined. The error message returned by <strong>SQLGetDiagRec</strong> in the <em>MessageText</em> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <em>StatementHandle</em>. The function was called, and before it completed execution, <strong>SQLCancel</strong> or <strong>SQLCancelHandle</strong> was called on the <em>StatementHandle</em>. Then the function was called again on the <em>StatementHandle</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The function was called, and before it completed execution, <strong>SQLCancel</strong> or <strong>SQLCancelHandle</strong> was called on the <em>StatementHandle</em> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null</td>
<td>(DM) <em>StatementText</em> was a null pointer.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| HY010  | Function sequence error              | (DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLExecDirect function was called.  
(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.  
(DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called.  
(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. |
| HY013  | Memory management error              | The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions. |
| HY090  | Invalid string or buffer length      | (DM) The argument TextLength was less than or equal to 0 but not equal to SQL_NTS.  
A parameter value, set with SQLBindParameter, was a null pointer, and the parameter length value was not 0, SQL_NULL_DATA, SQL_DATA_AT_EXEC, SQL_DEFAULT_PARAM, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.  
A parameter value, set with SQLBindParameter, was not a null pointer; the C data type was SQL_C_BINARY or SQL_C_CHAR; and the parameter length value was less than 0 but was not SQL_NTS, SQL_NULL_DATA, SQL_DATA_AT_EXEC, SQL_DEFAULT_PARAM, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.  
A parameter length value bound by SQLBindParameter was set to SQL_DATA_AT_EXEC; the SQL type was either SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long data source–specific data type; and the SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo was “Y”. |
<p>| HY105  | Invalid parameter type               | The value specified for the argument InputOutputType in SQLBindParameter was SQL_PARAM_OUTPUT, and the parameter was an input parameter. |
| HY109  | Invalid cursor position              | *StatementText contained a positioned update or delete statement, and the cursor was positioned (by SQLSetPos or |</p>
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes was not supported by the driver or data source. The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><strong>SQLCompleteAsync</strong> has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, <strong>SQLCompleteAsync</strong> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

**Comments**

The application calls **SQLExecDirect** to send an SQL statement to the data source. For more information about direct execution, see Direct Execution. The driver modifies the statement to use the form of SQL used by the data source and then submits it to the data source. In particular, the driver modifies the escape sequences used to define certain features in SQL. For the syntax of escape sequences, see Escape Sequences in ODBC.
The application can include one or more parameter markers in the SQL statement. To include a parameter marker, the application embeds a question mark (?) into the SQL statement at the appropriate position. For information about parameters, see Statement Parameters.

If the SQL statement is a SELECT statement and if the application called SQLSetCursorName to associate a cursor with a statement, then the driver uses the specified cursor. Otherwise, the driver generates a cursor name.

If the data source is in manual-commit mode (requiring explicit transaction initiation) and a transaction has not already been initiated, the driver initiates a transaction before it sends the SQL statement. For more information, see Manual-Commit Mode.

If an application uses SQLExecDirect to submit a COMMIT or ROLLBACK statement, it will not be interoperable between DBMS products. To commit or roll back a transaction, an application calls SQLEndTran.

If SQLExecDirect encounters a data-at-execution parameter, it returns SQL_NEED_DATA. The application sends the data using SQLParamData and SQLPutData. See SQLBindParameter, SQLParamData, SQLPutData, and Sending Long Data.

If SQLExecDirect executes a searched update, insert, or delete statement that does not affect any rows at the data source, the call to SQLExecDirect returns SQL_NO_DATA.

If the value of the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1 and the SQL statement contains at least one parameter marker, SQLExecDirect will execute the SQL statement once for each set of parameter values from the arrays pointed to by the ParameterValuePointer argument in the call to SQLBindParameter. For more information, see Arrays of Parameter Values.

If bookmarks are turned on and a query is executed that cannot support bookmarks, the driver should attempt to coerce the environment to one that supports bookmarks by changing an attribute value and returning SQLSTATE 01S02 (Option value changed). If the attribute cannot be changed, the driver should return SQLSTATE HY024 (Invalid attribute value).

**Note**

When using connection pooling, an application must not execute SQL statements that change the database or the context of the database, such as the USE database statement in SQL Server, which changes the catalog used by a data source.

**Code Example**

See SQLBindCol, SQLGetData, and Sample ODBC Program.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Task</td>
<td>Function</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Executing a commit or rollback operation</td>
<td>SQLEndTran Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
<tr>
<td>Fetching multiple rows of data</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Returning a cursor name</td>
<td>SQLGetCursorName Function</td>
</tr>
<tr>
<td>Fetching part or all of a column of data</td>
<td>SQLGetData Function</td>
</tr>
<tr>
<td>Returning the next parameter to send data for</td>
<td>SQLParamData Function</td>
</tr>
<tr>
<td>Preparing a statement for execution</td>
<td>SQLPrepare Function</td>
</tr>
<tr>
<td>Sending parameter data at execution time</td>
<td>SQLPutData Function</td>
</tr>
<tr>
<td>Setting a cursor name</td>
<td>SQLSetCursorName Function</td>
</tr>
<tr>
<td>Setting a statement attribute</td>
<td>SQLSetStmtAttr Function</td>
</tr>
</tbody>
</table>

**See Also**

- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft

# SQLExecute Function

## Conformance

Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

## Summary

`SQLExecute` executes a prepared statement, using the current values of the parameter marker variables if any parameter markers exist in the statement.

## Syntax

```c
SQLRETURN SQLExecute(
    SQLHSTMT StatementHandle);
```
Arguments

StatementHandle
  [Input] Statement handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, SQL_NO_DATA, SQL_INVALID_HANDLE, or SQL_PARAM_DATA_AVAILABLE.

Diagnostics

When SQLExecute returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLExecute and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01001</td>
<td>Cursor operation conflict</td>
<td>The prepared statement associated with the StatementHandle contained a positioned update or delete statement, and no rows or more than one row were updated or deleted. (For more information about updates to more than one row, see the description of the SQL_ATTR_SIMULATE_CURSOR Attribute in SQLSetStmtAttr.) (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01003</td>
<td>NULL value eliminated in set function</td>
<td>The prepared statement associated with StatementHandle contained a set function (such as AVG, MAX, MIN, and so on), but not the COUNT set function, and NULL argument values were eliminated before the function was applied. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>String or binary data returned for an output parameter resulted in the truncation of nonblank character or non-NULL binary data. If it was a string value, it was right-truncated. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01006</td>
<td>Privilege not revoked</td>
<td>The prepared statement associated with the StatementHandle</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>01007</td>
<td>Privilege not granted. The prepared statement associated with the <code>StatementHandle</code> was a <strong>REVOKE</strong> statement, and the user did not have the specified privilege. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
<td></td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed. A specified statement attribute was invalid because of implementation working conditions, so a similar value was temporarily substituted. (<strong>SQLGetStmtAttr</strong> can be called to determine what the temporarily substituted value is.) The substitute value is valid for the <code>StatementHandle</code> until the cursor is closed, at which point the statement attribute reverts to its previous value. The statement attributes that can be changed are: <code>SQL_ATTR_CONCURRENCY</code>, <code>SQL_ATTR_CURSOR_TYPE</code>, <code>SQL_ATTR_KEYSET_SIZE</code>, <code>SQL_ATTR_MAX_LENGTH</code>, <code>SQL_ATTR_MAX_ROWS</code>, <code>SQL_ATTR_QUERY_TIMEOUT</code>, and <code>SQL_ATTR_SIMULATE_CURSOR</code>. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
<td></td>
</tr>
<tr>
<td>01S07</td>
<td>Fractional truncation. The data returned for an input/output or output parameter was truncated such that the fractional part of a numeric data type was truncated or the fractional portion of the time component of a time, timestamp, or interval data type was truncated. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
<td></td>
</tr>
<tr>
<td>07002</td>
<td>COUNT field incorrect. The number of parameters specified in <strong>SQLBindParameter</strong> was less than the number of parameters in the SQL statement contained in <code>*StatementText</code>. <strong>SQLBindParameter</strong> was called with <code>ParameterValuePtr</code> set to a null pointer, <code>StrLen_or_IndPtr</code> not set to SQL_NULL_DATA or SQL_DATA_AT_EXEC, and <code>InputOutputType</code> not set to SQL_PARAM_OUTPUT, so that the number of parameters specified in <strong>SQLBindParameter</strong> was greater than the number of parameters in the SQL statement contained in <code>*StatementText</code>.</td>
<td></td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation. The data value identified by the <code>ValueType</code> argument in <strong>SQLBindParameter</strong> for the bound parameter could not be converted to the data type identified by the <code>ParameterType</code> argument in <strong>SQLBindParameter</strong>. The data value returned for a parameter bound as SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT could not be converted to the data type identified by the <code>ValueType</code> argument in <strong>SQLBindParameter</strong>. (If the data values for one or more rows could not be converted but one or more rows were successfully returned, this function returns SQL_SUCCESS_WITH_INFO.)</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 07007  | Restricted parameter value violation             | The parameter type SQL_PARAM_INPUT_OUTPUT_STREAM is only used for a parameter that sends and receives data in parts. An input bound buffer is not allowed for this parameter type.
<p>|        |                                                  | This error will occur when the parameter type is SQL_PARAM_INPUT_OUTPUT, and when the *StrLen_or_IndPtr specified in SQLBindParameter is not equal to SQL_NULL_DATA, SQL_DEFAULT_PARAM, SQL_LEN_DATA_AT_EXEC(len), or SQL_DATA_AT_EXEC. |
| 07S01  | Invalid use of default parameter                 | A parameter value, set with SQLBindParameter, was SQL_DEFAULT_PARAM, and the corresponding parameter was not a parameter for an ODBC canonical procedure invocation. |
| 08S01  | Communication link failure                       | The communication link between the driver and the data source to which the driver was connected failed before the function completed processing. |
| 21S02  | Degree of derived table does not match column list | The prepared statement associated with the StatementHandle contained a CREATE VIEW statement, and the unqualified column list (the number of columns specified for the view in the column-identifier arguments of the SQL statement) contained more names than the number of columns in the derived table defined by the query-specification argument of the SQL statement. |
| 22001  | String data, right truncation                    | The assignment of a character or binary value to a column resulted in the truncation of nonblank (character) or non-null (binary) characters or bytes. |
| 22002  | Indicator variable required but not supplied     | NULL data was bound to an output parameter whose StrLen_or_IndPtr set by SQLBindParameter was a null pointer. |
| 22003  | Numeric value out of range                       | The prepared statement associated with the StatementHandle contained a bound numeric parameter, and the parameter value caused the whole (as opposed to fractional) part of the number to be truncated when assigned to the associated table column. |
|        |                                                  | Returning a numeric value (as numeric or string) for one or more input/output or output parameters would have caused the whole (as opposed to fractional) part of the number to be truncated. |
| 22007  | Invalid datetime format                          | The prepared statement associated with the StatementHandle contained an SQL statement that contained a date, time, or timestamp structure as a bound parameter, and the parameter was, respectively, an invalid date, time, or timestamp. |
|        |                                                  | An input/output or output parameter was bound to a date, time, or timestamp C structure, and a value in the returned parameter was, respectively, an invalid date, time, or |</p>
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>22008</td>
<td>Datetime field overflow</td>
<td>The prepared statement associated with the <code>StatementHandle</code> contained an SQL statement that contained a datetime expression that, when computed, resulted in a date, time, or timestamp structure that was invalid. A datetime expression computed for an input/output or output parameter resulted in a date, time, or timestamp C structure that was invalid.</td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
<td>The prepared statement associated with the <code>StatementHandle</code> contained an arithmetic expression that caused division by zero. An arithmetic expression calculated for an input/output or output parameter resulted in division by zero.</td>
</tr>
<tr>
<td>22015</td>
<td>Interval field overflow</td>
<td><em>StatementText</em> contained an exact numeric or interval parameter that, when converted to an interval SQL data type, caused a loss of significant digits. <em>StatementText</em> contained an interval parameter with more than one field that, when converted to a numeric data type in a column, had no representation in the numeric data type. <em>StatementText</em> contained parameter data that was assigned to an interval SQL type, and there was no representation of the value of the C type in the interval SQL type. Assigning an input/output or output parameter that was an exact numeric or interval SQL type to an interval C type caused a loss of significant digits. When an input/output or output parameter was assigned to an interval C structure, there was no representation of the data in the interval data structure.</td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast specification</td>
<td><em>StatementText</em> contained a C type that was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type. When an input/output or output parameter was returned, the SQL type was an exact or approximate numeric, a datetime, or an interval data type; the C type was SQL_C_CHAR; and the value in the column was not a valid literal of the bound SQL type.</td>
</tr>
<tr>
<td>22019</td>
<td>Invalid escape character</td>
<td>The prepared statement associated with <code>StatementHandle</code> contained a <strong>LIKE</strong> predicate with an <strong>ESCAPE</strong> in the <strong>WHERE</strong> clause, and the length of the escape character following <strong>ESCAPE</strong> was not equal to 1.</td>
</tr>
<tr>
<td>Status Code</td>
<td>Error Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>22025</td>
<td>Invalid escape sequence</td>
<td></td>
</tr>
<tr>
<td>23000</td>
<td>Integrity constraint violation</td>
<td></td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td></td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td></td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td></td>
</tr>
<tr>
<td>42000</td>
<td>Syntax error or access violation</td>
<td></td>
</tr>
<tr>
<td>44000</td>
<td>WITH CHECK OPTION violation</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td></td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation</td>
<td></td>
</tr>
</tbody>
</table>

22025 Invalid escape sequence: The prepared statement associated with `StatementHandle` contained "LIKE pattern value ESCAPE escape character" in the WHERE clause, and the character following the escape character in the pattern value was not one of "%" or "_".

23000 Integrity constraint violation: The prepared statement associated with the `StatementHandle` contained a parameter. The parameter value was NULL for a column defined as NOT NULL in the associated table column, a duplicate value was supplied for a column constrained to contain only unique values, or some other integrity constraint was violated.

24000 Invalid cursor state: A cursor was positioned on the `StatementHandle` by `SQLFetch` or `SQLFetchScroll`. This error is returned by the Driver Manager if `SQLFetch` or `SQLFetchScroll` has not returned SQL_NO_DATA, and is returned by the driver if `SQLFetch` or `SQLFetchScroll` has returned SQL_NO_DATA. A cursor was open on the `StatementHandle`.

40001 Serialization failure: The transaction was rolled back due to a resource deadlock with another transaction.

40003 Statement completion unknown: The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.

42000 Syntax error or access violation: The user did not have permission to execute the prepared statement associated with the `StatementHandle`.

44000 WITH CHECK OPTION violation: The prepared statement associated with `StatementHandle` contained an INSERT statement performed on a viewed table or a table derived from the viewed table that was created by specifying WITH CHECK OPTION, such that one or more rows affected by the INSERT statement will no longer be present in the viewed table.

The prepared statement associated with the `StatementHandle` contained an UPDATE statement performed on a viewed table or a table derived from the viewed table that was created by specifying WITH CHECK OPTION, such that one or more rows affected by the UPDATE statement will no longer be present in the viewed table.

HY000 General error: An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by `SQLGetDiagRec` in the *MessageText buffer describes the error and its cause.

HY001 Memory allocation: The driver was unable to allocate memory required to support
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
</tr>
<tr>
<td></td>
<td>Asynchronous processing was enabled for the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code>. Then the function was called again on the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
</tr>
<tr>
<td></td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when the <code>SQLExecute</code> function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for the <code>StatementHandle</code> and returned <code>SQL_PARAM_DATA_AVAILABLE</code>. This function was called before data was retrieved for all streamed parameters. (DM) An asynchronously executing function (not this one) was called for the <code>StatementHandle</code> and was still executing when this function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for the <code>StatementHandle</code> and returned <code>SQL_NEED_DATA</code>. This function was called before data was sent for all data-at-execution parameters or columns. (DM) The <code>StatementHandle</code> was not prepared.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
</tr>
<tr>
<td></td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
</tr>
</tbody>
</table>
|            | A parameter value, set with `SQLBindParameter`, was a null pointer, and the parameter length value was not 0, `SQL_NULL_DATA`, `SQL_DATA_AT_EXEC`, `SQL_DEFAULT_PARAM`, or less than or equal to `SQL_LEN_DATA_AT_EXEC_OFFSET`. A parameter value, set with `SQLBindParameter`, was not a null pointer; the C data type was `SQL_C_BINARY` or `SQL_C_CHAR`; and the parameter length value was less than 0 but was not `SQL_NTS`, `SQL_NULL_DATA`, `SQL_DEFAULT_PARAM`, or `SQL_DATA_AT_EXEC`, or less than or equal to `SQL_LEN_DATA_AT_EXEC_OFFSET`. A parameter length value bound by `SQLBindParameter` was set to `SQL_DATA_AT_EXEC`; the SQL type was either `SQL_LONGVARCHAR`, `SQL_LONGVARBINARY`, or a long data source-specific data type; and the
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY105</td>
<td>Invalid parameter type</td>
<td>The value specified for the argument InputOutputType in SQLBindParameter was SQL_PARAM_OUTPUT, and the parameter was an input parameter.</td>
</tr>
<tr>
<td>HY109</td>
<td>Invalid cursor position</td>
<td>The prepared statement was a positioned update or delete statement, and the cursor was positioned (by SQLSetPos or SQLFetchScroll) on a row that had been deleted or could not be fetched.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes was not supported by the driver or data source. The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td>SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>
**SQLExecute** can return any SQLSTATE that can be returned by **SQLPrepare**, based on when the data source evaluates the SQL statement associated with the statement.

**Comments**

**SQLExecute** executes a statement prepared by **SQLPrepare**. After the application processes or discards the results from a call to **SQLExecute**, the application can call **SQLExecute** again with new parameter values. For more information about prepared execution, see Prepared Execution.

To execute a **SELECT** statement more than once, the application must call **SQLCloseCursor** before reexecuting the **SELECT** statement.

If the data source is in manual-commit mode (requiring explicit transaction initiation) and a transaction has not already been initiated, the driver initiates a transaction before it sends the SQL statement. For more information, see Transactions.

If an application uses **SQLPrepare** to prepare and **SQLExecute** to submit a **COMMIT** or **ROLLBACK** statement, it will not be interoperable between DBMS products. To commit or roll back a transaction, call **SQLEndTran**.

If **SQLExecute** encounters a data-at-execution parameter, it returns SQL_NEED_DATA. The application sends the data using **SQLParamData** and **SQLPutData**. See **SQLBindParameter**, **SQLParamData**, **SQLPutData**, and Sending Long Data.

If **SQLExecute** executes a searched update, insert, or delete statement that does not affect any rows at the data source, the call to **SQLExecute** returns SQL_NO_DATA.

If the value of the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1 and the SQL statement contains at least one parameter marker, **SQLExecute** executes the SQL statement once for each set of parameter values in the arrays pointed to by the *ParameterValuePtr* argument in the calls to **SQLBindParameter**. For more information, see Arrays of Parameter Values.

If bookmarks are enabled and a query is executed that cannot support bookmarks, the driver should attempt to coerce the environment to one that supports bookmarks by changing an attribute value and returning SQLSTATE 01S02 (Option value changed). If the attribute cannot be changed, the driver should return SQLSTATE HY024 (Invalid attribute value).

**Note**

When using connection pooling, an application must not execute SQL statements that change the database or the context of the database, such as the **USE** database statement in SQL Server, which changes the catalog used by a data source.

**Code Example**

See **SQLBindParameter**, **SQLBulkOperations**, **SQLPutData**, and **SQLSetPos**.

**Related Functions**
<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Closing the cursor</td>
<td>SQLCloseCursor Function</td>
</tr>
<tr>
<td>Executing a commit or rollback operation</td>
<td>SQLEndTran Function</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect Function</td>
</tr>
<tr>
<td>Fetching multiple rows of data</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Freeing a statement handle</td>
<td>SQLFreeStmt Function</td>
</tr>
<tr>
<td>Returning a cursor name</td>
<td>SQLGetCursorName Function</td>
</tr>
<tr>
<td>Fetching part or all of a column of data</td>
<td>SQLGetData Function</td>
</tr>
<tr>
<td>Returning the next parameter to send data for</td>
<td>SQLParamData Function</td>
</tr>
<tr>
<td>Preparing a statement for execution</td>
<td>SQLPrepare Function</td>
</tr>
<tr>
<td>Sending parameter data at execution time</td>
<td>SQLPutData Function</td>
</tr>
<tr>
<td>Setting a cursor name</td>
<td>SQLSetCursorName Function</td>
</tr>
<tr>
<td>Setting a statement attribute</td>
<td>SQLSetStmtAttr Function</td>
</tr>
</tbody>
</table>

**See Also**

ODBC API Reference  
ODBC Header Files

© 2016 Microsoft

**SQLExtendedFetch Function**

**Conformance**

Version Introduced: ODBC 1.0  
Standards Compliance: Deprecated

**Summary**

SQLExtendedFetch fetches the specified rowset of data from the result set and returns data for
all bound columns. Rowsets can be specified at an absolute or relative position or by bookmark.

**Note**

In ODBC 3.x, **SQLExtendedFetch** has been replaced by **SQLFetchScroll**. ODBC 3.x applications should not call **SQLExtendedFetch**; instead they should call **SQLFetchScroll**. The Driver Manager maps **SQLFetchScroll** to **SQLExtendedFetch** when working with an ODBC 2.x driver. ODBC 3.x drivers should support **SQLExtendedFetch** if they want to work with ODBC 2.x applications that call it. For more information, see "Comments" and Block Cursors, Scrollable Cursors, and Backward Compatibility in Appendix G: Driver Guidelines for Backward Compatibility.

**Syntax**

```c
SQLRETURN SQLExtendedFetch(
    SQLHSTMT    StatementHandle,
    SQLUSMALLINT FetchOrientation,
    SQLLEN      FetchOffset,
    SQLULEN *  RowCountPtr,
    SQLUSMALLINT * RowStatusArray);
```

**Arguments**

*StatementHandle*

[Input] Statement handle.

*FetchOrientation*

[Input] Type of fetch. This is the same as `FetchOrientation` in **SQLFetchScroll**.

*FetchOffset*

[Input] Number of the row to fetch. This is the same as `FetchOffset` in **SQLFetchScroll**, with one exception. When `FetchOrientation` is SQL_FETCH_BOOKMARK, `FetchOffset` is a fixed-length bookmark, not an offset from a bookmark. In other words, **SQLExtendedFetch** retrieves the bookmark from this argument, not the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute. It does not support variable-length bookmarks and does not support fetching a rowset at an offset (other than 0) from a bookmark.

*RowCountPtr*

[Output] Pointer to a buffer in which to return the number of rows actually fetched. This buffer is used in the same manner as the buffer specified by the SQL_ATTR_ROWS_FETCHED_PTR statement attribute. This buffer is used only by **SQLExtendedFetch**. It is not used by **SQLFetch** or **SQLFetchScroll**.

*RowStatusArray*

[Output] Pointer to an array in which to return the status of each row. This array is used in the same manner as the array specified by the SQL_ATTR_ROW_STATUS_PTR statement attribute.

However, the address of this array is not stored in the SQL_DESC_STATUS_ARRAY_PTR field in the IRD. Furthermore, this array is used only by **SQLExtendedFetch** and by **SQLBulkOperations** with an `Operation` of SQL_ADD or SQLSetPos when it is called after **SQLExtendedFetch**. It is not used by **SQLFetch** or **SQLFetchScroll**, and it is not used by
SQLBulkOperations or SQLSetPos when they are called after SQLFetch or SQLFetchScroll. It is also not used when SQLBulkOperations with an Operation of SQL_ADD is called before any fetch function is called. In other words, it is used only in statement state S7. It is not used in statement states S5 or S6. For more information, see Statement Transitions in Appendix B: ODBC State Transition Tables.

Applications should provide a valid pointer in the RowStatusArray argument; if not, the behavior of SQLExtendedFetch and the behavior of calls to SQLBulkOperations or SQLSetPos after a cursor has been positioned by SQLExtendedFetch are undefined.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLExtendedFetch returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLError. The following table lists the SQLSTATE values commonly returned by SQLExtendedFetch and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise. If an error occurs on a single column, SQLGetDiagField can be called with a DiagIdentifier of SQL_DIAG_COLUMN_NUMBER to determine the column the error occurred on; and SQLGetDiagField can be called with a DiagIdentifier of SQL_DIAG_ROW_NUMBER to determine the row containing that column.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>String or binary data returned for a column resulted in the truncation of nonblank character or non-NULL binary data. If it was a string value, it was right-truncated. If it was a numeric value, the fractional part of the number was truncated. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S01</td>
<td>Error in row</td>
<td>An error occurred while fetching one or more rows. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S06</td>
<td>Attempt to fetch before the result set returned the first rowset</td>
<td>The requested rowset overlapped the start of the result set when the current position was beyond the first row, and either FetchOrientation was SQL_PRIOR or FetchOrientation was SQL_RELATIVE with a negative FetchOffset whose absolute value was less than or equal to the current SQL_ROWSET_SIZE. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S07</td>
<td>Fractional</td>
<td>The data returned for a column was truncated. For numeric data</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td></td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td></td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td></td>
</tr>
<tr>
<td>22002</td>
<td>Indicator variable required but not supplied</td>
<td></td>
</tr>
<tr>
<td>22003</td>
<td>Numeric value out of range</td>
<td></td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format</td>
<td></td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
<td></td>
</tr>
<tr>
<td>22015</td>
<td>Interval field overflow</td>
<td></td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast</td>
<td></td>
</tr>
</tbody>
</table>

For more information, see Guidelines for Interval and Numeric Data Types in Appendix D: Data Types.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Description</th>
</tr>
</thead>
</table>
| specification | character data type; and the value in the column was not a valid literal of the bound C type.  
(Function returns SQL_SUCCESS_WITH_INFO.) |
| 24000 | Invalid cursor state  
The *StatementHandle* was in an executed state, but no result set was associated with the *StatementHandle*. |
| HY000 | General error  
An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLError in the *MessageText* buffer describes the error and its cause. |
| HY001 | Memory allocation error  
The driver was unable to allocate memory required to support execution or completion of the function. |
| HY008 | Operation canceled  
Asynchronous processing was enabled for the *StatementHandle*. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the *StatementHandle*, and then the function was called again on the *StatementHandle*.  
The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the *StatementHandle* from a different thread in a multithread application. |
| HY010 | Function sequence error  
(DM) An asynchronously executing function was called for the connection handle that is associated with the *StatementHandle*. This asynchronous function was still executing when the SQLExtendedFetch function was called.  
(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the *StatementHandle* and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.  
(DM) The specified *StatementHandle* was not in an executed state. The function was called without first calling SQLExecDirect, SQLExecute, or a catalog function.  
(DM) An asynchronously executing function (not this one) was called for the *StatementHandle* and was still executing when this function was called.  
(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.  
(DM) SQLExtendedFetch was called for the *StatementHandle* after SQLFetch or SQLFetchScroll was called and before SQLFreeStmt was called with the SQL_CLOSE option.  
(DM) SQLBulkOperations was called for a statement before SQLFetch, SQLFetchScroll, or SQLExtendedFetch was called. |
called, and then **SQLExtendedFetch** was called before **SQLFreeStmt** was called with the SQL_CLOSE option.

<table>
<thead>
<tr>
<th>Code</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY106</td>
<td>Fetch type out of range</td>
<td>(DM) The value specified for the argument <em>FetchOrientation</em> was invalid. (See &quot;Comments.&quot;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The argument <em>FetchOrientation</em> was SQL_FETCH_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value of the SQL_CURSOR_TYPE statement option was SQL_CURSOR_FORWARD_ONLY, and the value of argument <em>FetchOrientation</em> was not SQL_FETCH_NEXT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The argument <em>FetchOrientation</em> was SQL_FETCH_RESUME.</td>
</tr>
<tr>
<td>HY107</td>
<td>Row value out of range</td>
<td>The value specified with the SQL_CURSOR_TYPE statement option was SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the SQL_KEYSET_SIZE statement attribute was greater than 0 and less than the value specified with the SQL_ROWSET_SIZE statement attribute.</td>
</tr>
<tr>
<td>HY111</td>
<td>Invalid bookmark value</td>
<td>The argument <em>FetchOrientation</em> was SQL_FETCH_BOOKMARK, and the bookmark specified in the <em>FetchOffset</em> argument was not valid.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) For more information about suspended state, see <strong>SQLEndTran Function</strong>.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>Driver or data source does not support the specified fetch type. The driver or data source does not support the conversion specified by the combination of the <em>TargetType</em> in <strong>SQLBindCol</strong> and the SQL data type of the corresponding column. This error applies only when the SQL data type of the column was mapped to a driver-specific SQL data type.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the result set. The timeout period is set through <strong>SQLSetStmtOption</strong>, SQL_QUERY_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <strong>SQLSetConnectAttr</strong>, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
</tbody>
</table>
Notes

Driver does not support this function

(DM) The driver associated with the StatementHandle does not support the function.

Comments

The behavior of SQLExtendedFetch is identical to that of SQLFetchScroll, with the following exceptions:

- SQLExtendedFetch and SQLFetchScroll use different methods to return the number of rows fetched. SQLExtendedFetch returns the number of rows fetched in *RowCountPtr; SQLFetchScroll returns the number of rows fetched directly to the buffer pointed to by SQL_ATTR_ROWS_FETCHED_PTR. For more information, see the RowCountPtr argument.

- SQLExtendedFetch and SQLFetchScroll return the status of each row in different arrays. For more information, see the RowStatusArray argument.

- SQLExtendedFetch and SQLFetchScroll use different methods to retrieve the bookmark when FetchOrientation is SQL_FETCH_BOOKMARK. SQLExtendedFetch does not support variable-length bookmarks or fetching rowsets at an offset other than 0 from a bookmark. For more information, see the FetchOffset argument.

- SQLExtendedFetch and SQLFetchScroll use different rowset sizes. SQLExtendedFetch uses the value of the SQL_ROWSET_SIZE statement attribute, and SQLFetchScroll uses the value of the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.

- SQLExtendedFetch has slightly different error handling semantics than SQLFetchScroll. For more information, see "Error Handling" in the "Comments" section of SQLFetchScroll.

- SQLExtendedFetch does not support binding offsets (the SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute).

- Calls to SQLExtendedFetch cannot be mixed with calls to SQLFetch or SQLFetchScroll, and if SQLBulkOperations is called before any fetch function is called, SQLExtendedFetch cannot be called until the cursor is closed and reopened. That is, SQLExtendedFetch can be called only in statement state S7. For more information, see Statement Transitions in Appendix B: ODBC State Transition Tables.

When an application calls SQLFetchScroll while using an ODBC 2.x driver, the Driver Manager maps this call to SQLExtendedFetch. For more information, see "SQLFetchScroll and ODBC 2.x Drivers" in SQLFetchScroll.

In ODBC 2.x, SQLExtendedFetch was called to fetch multiple rows and SQLFetch was called to fetch a single row. In ODBC 3.x, on the other hand, SQLFetch can be called to fetch multiple rows.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### SQLFetch Function

#### Conformance

Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

#### Summary

**SQLFetch** fetches the next rowset of data from the result set and returns data for all bound columns.

#### Syntax

```c
SQLRETURN SQLFetch(
    SQLHSTMT    StatementHandle);
```
Arguments

StatementHandle
  [Input] Statement handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLFetch returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec Function with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values typically returned by SQLFetch and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise. If an error occurs on a single column, SQLGetDiagField can be called with a DiagIdentifier of SQL_DIAG_COLUMN_NUMBER to determine the column the error occurred on; and SQLGetDiagField can be called with a DiagIdentifier of SQL_DIAG_ROW_NUMBER to determine the row that contains that column.

For all those SQLSTATEs that can return SQL_SUCCESS_WITH_INFO or SQL_ERROR (except 01xxx SQLSTATEs), SQL_SUCCESS_WITH_INFO is returned if an error occurs on one or more, but not all, rows of a multirow operation, and SQL_ERROR is returned if an error occurs on a single-row operation.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>String or binary data returned for a column resulted in the truncation of nonblank character or non-NULL binary data. If it was a string value, it was right-truncated.</td>
</tr>
<tr>
<td>01S01</td>
<td>Error in row</td>
<td>An error occurred while fetching one or more rows. (If this SQLSTATE is returned when an ODBC 3.x application is working with an ODBC 2.x driver, it can be ignored.)</td>
</tr>
<tr>
<td>01S07</td>
<td>Fractional truncation</td>
<td>The data returned for a column was truncated. For numeric data types, the fractional part of the number was truncated. For time, timestamp, and interval data types that contain a time component, the fractional part of the time was truncated.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>The data value of a column in the result set could not be converted to the data type specified by TargetType in SQLBindCol. Column 0 was bound with a data type of SQL_C_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE. Column 0 was bound with a data type of SQL_C_VARCHAR, and the SQL_ATTR_USE_BOOKMARKS statement attribute was not set to SQL_UB_VARIABLE.</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>The driver was an ODBC 2.x driver that does not support SQLExtendedFetch, and a column number specified in the binding for a column was 0. Column 0 was bound, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.</td>
</tr>
<tr>
<td>08001</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>22001</td>
<td>String data, right truncated</td>
<td>A variable-length bookmark returned for a column was truncated.</td>
</tr>
<tr>
<td>22002</td>
<td>Indicator variable required but not supplied</td>
<td>NULL data was fetched into a column whose StrLen_or_IndPtr set by SQLBindCol (or SQL_DESC_INDICATOR_PTR set by SQLSetDescField or SQLSetDescRec) was a null pointer.</td>
</tr>
<tr>
<td>22003</td>
<td>Numeric value out of range</td>
<td>Returning the numeric value as numeric or string for one or more bound columns would have caused the whole (as opposed to fractional) part of the number to be truncated. For more information, see Converting Data from SQL to C Data Types in Appendix D: Data Types.</td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format</td>
<td>A character column in the result set was bound to a date, time, or timestamp C structure, and a value in the column was, respectively, an invalid date, time, or timestamp.</td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
<td>A value from an arithmetic expression was returned, which resulted in division by zero.</td>
</tr>
<tr>
<td>22015</td>
<td>Interval field overflow</td>
<td>Assigning from an exact numeric or interval SQL type to an interval C type caused a loss of significant digits in the leading field. When fetching data to an interval C type, there was no representation of the value of the SQL type in the interval C type.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Error Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast specification</td>
<td>A character column in the result set was bound to a character C buffer, and the column contained a character for which there was no representation in the character set of the buffer. The C type was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>The <em>StatementHandle</em> was in an executed state but no result set was associated with the <em>StatementHandle</em>.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction in which the fetch was executed was terminated to prevent deadlock.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <em>SQLGetDiagRec</em> in the <em>MessageText</em> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <em>StatementHandle</em>. The <em>SQLFetch</em> function was called, and before it completed execution, <em>SQLCancel</em> or <em>SQLCancelHandle</em> was called on the <em>StatementHandle</em>. Then the <em>SQLFetch</em> function was called again on the <em>StatementHandle</em>. Or, the <em>SQLFetch</em> function was called, and before it completed execution, <em>SQLCancel</em> or <em>SQLCancelHandle</em> was called on the <em>StatementHandle</em> from a different thread in a multithread application.</td>
</tr>
</tbody>
</table>
| HY010  | Function sequence error                               | (DM) An asynchronously executing function was called for the connection handle that is associated with the *StatementHandle*. This asynchronous function was still executing when the *SQLFetch* function was called. (DM) *SQLExecute*, *SQLExecDirect*, or *SQLMoreResults* was called for the *StatementHandle* and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. (DM) The specified *StatementHandle* was not in an executed state. The function was called without first calling *SQLExecDirect*, *SQLExecute* or a catalog function. (DM) An asynchronously executing function (not this one)
was called for the `StatementHandle` and was still executing when this function was called.

(DM) `SQLExecute`, `SQLExecDirect`, `SQLBulkOperations`, or `SQLSetPos` was called for the `StatementHandle` and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

(DM) `SQLFetch` was called for the `StatementHandle` after `SQLExtendedFetch` was called and before `SQLFreeStmt` with the SQL_CLOSE option was called.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>The SQL_ATTR_USE_BOOKMARK statement attribute was set to SQL_UB_VARIABLE, and column 0 was bound to a buffer whose length was not equal to the maximum length for the bookmark for this result set. (This length is available in the SQL_DESC_OCTET_LENGTH field of the IRD and can be obtained by calling <code>SQLDescribeCol</code>, <code>SQLColAttribute</code>, or <code>SQLGetDescField</code>.)</td>
</tr>
<tr>
<td>HY107</td>
<td>Row value out of range</td>
<td>The value specified with the SQL_ATTR_CURSOR_TYPE statement attribute was SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the SQL_ATTR_KEYSET_SIZE statement attribute was greater than 0 and less than the value specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see <code>SQLEndTran Function</code>.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The driver or data source does not support the conversion specified by the combination of the <code>TargetType</code> in <code>SQLBindCol</code> and the SQL data type of the corresponding column.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the requested result set. The timeout period is set through <code>SQLSetStmtAttr</code>, SQL_ATTR_QUERY_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not</td>
<td>(DM) The driver associated with the <code>StatementHandle</code> does</td>
</tr>
</tbody>
</table>
Comments

**SQLFetch** returns the next rowset in the result set. It can be called only while a result set exists: that is, after a call that creates a result set and before the cursor over that result set is closed. If any columns are bound, it returns the data in those columns. If the application has specified a pointer to a row status array or a buffer in which to return the number of rows fetched, **SQLFetch** also returns this information. Calls to **SQLFetch** can be mixed with calls to **SQLFetchScroll** but cannot be mixed with calls to **SQLExtendedFetch**. For more information, see Fetching a Row of Data.

If an ODBC 3.x application works with an ODBC 2.x driver, the Driver Manager maps **SQLFetch** calls to **SQLExtendedFetch** for an ODBC 2.x driver that supports **SQLExtendedFetch**. If the ODBC 2.x driver does not support **SQLExtendedFetch**, the Driver Manager maps **SQLFetch** calls to **SQLFetch** in the ODBC 2.x driver, which can fetch only a single row.

For more information, see Block Cursors, Scrollable Cursors, and Backward Compatibility in Appendix G: Driver Guidelines for Backward Compatibility.

Positioning the Cursor

When the result set is created, the cursor is positioned before the start of the result set. **SQLFetch** fetches the next rowset. It is equivalent to calling **SQLFetchScroll** with **FetchOrientation** set to SQL_FETCH_NEXT. For more information about cursors, see Cursors and Block Cursors.

The SQL_ATTR_ROW_ARRAY_SIZE statement attribute specifies the number of rows in the rowset. If the rowset being fetched by **SQLFetch** overlaps the end of the result set, **SQLFetch** returns a partial rowset. That is, if S + R – 1 is greater than L, where S is the starting row of the rowset being fetched, R is the rowset size, and L is the last row in the result set, then only the first L – S + 1 rows of the rowset are valid. The remaining rows are empty and have a status of SQL_ROW_NOROW.

After **SQLFetch** returns, the current row is the first row of the rowset.

The rules listed in the following table describe cursor positioning after a call to **SQLFetch**, based on the conditions listed in the second table in this section.

<table>
<thead>
<tr>
<th>Condition</th>
<th>First row of new rowset</th>
</tr>
</thead>
<tbody>
<tr>
<td>support this function</td>
<td>not support the function.</td>
</tr>
</tbody>
</table>
Before start

\[ CurrRowsetStart \leq LastResultRow - RowsetSize[1] \]

\[ CurrRowsetStart > LastResultRow - RowsetSize[1] \]

After end

After end

[1] If the rowset size is changed between fetches, this is the rowset size that was used with the previous fetch.

[2] If the rowset size is changed between fetches, this is the rowset size that was used with the new fetch.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before start</td>
<td>The block cursor is positioned before the start of the result set. If the first row of the new rowset is before the start of the result set, <code>SQLFetch</code> returns SQL_NO_DATA.</td>
</tr>
<tr>
<td>After end</td>
<td>The block cursor is positioned after the end of the result set. If the first row of the new rowset is after the end of the result set, <code>SQLFetch</code> returns SQL_NO_DATA.</td>
</tr>
<tr>
<td><code>CurrRowsetStart</code></td>
<td>The number of the first row in the current rowset.</td>
</tr>
<tr>
<td><code>LastResultRow</code></td>
<td>The number of the last row in the result set.</td>
</tr>
<tr>
<td><code>RowsetSize</code></td>
<td>The rowset size.</td>
</tr>
</tbody>
</table>

For example, suppose a result set has 100 rows and the rowset size is 5. The following table shows the rowset and return code returned by `SQLFetch` for different starting positions.

<table>
<thead>
<tr>
<th>Current rowset</th>
<th>Return code</th>
<th>New rowset</th>
<th># of rows fetched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before start</td>
<td>SQL_SUCCESS</td>
<td>1 to 5</td>
<td>5</td>
</tr>
<tr>
<td>1 to 5</td>
<td>SQL_SUCCESS</td>
<td>6 to 10</td>
<td>5</td>
</tr>
<tr>
<td>52 to 56</td>
<td>SQL_SUCCESS</td>
<td>57 to 61</td>
<td>5</td>
</tr>
<tr>
<td>91 to 95</td>
<td>SQL_SUCCESS</td>
<td>96 to 100</td>
<td>5</td>
</tr>
<tr>
<td>93 to 97</td>
<td>SQL_SUCCESS</td>
<td>98 to 100. Rows 4 and 5 of the row status array are set to SQL_ROW_NOROW.</td>
<td>3</td>
</tr>
<tr>
<td>96 to 100</td>
<td>SQL_NO_DATA</td>
<td>None.</td>
<td>0</td>
</tr>
</tbody>
</table>
Returning Data in Bound Columns

As SQLFetch returns each row, it puts the data for each bound column in the buffer bound to that column. If no columns are bound, SQLFetch returns no data but does move the block cursor forward. The data can still be retrieved by using SQLGetData. If the cursor is a multihrow cursor (that is, the SQL_ATTR_ROW_ARRAY_SIZE is greater than 1), SQLGetData can be called only if SQL_GD_BLOCK is returned when SQLGetInfo is called with an InfoType of SQL_GETDATA_EXTENSIONS. (For more information, see SQLGetData.)

For each bound column in a row, SQLFetch does the following:

1. Sets the length/indicator buffer to SQL_NULL_DATA and proceeds to the next column if the data is NULL. If the data is NULL and no length/indicator buffer was bound, SQLFetch returns SQLSTATE 22002 (Indicator variable required but not supplied) for the row and proceeds to the next row. For information about how to determine the address of the length/indicator buffer, see "Buffer Addresses" in SQLBindCol.

   If the data for the column is not NULL, SQLFetch proceeds to step 2.

2. If the SQL_ATTR_MAX_LENGTH statement attribute is set to a nonzero value and the column contains character or binary data, the data is truncated to SQL_ATTR_MAX_LENGTH bytes.

   **Note**

   The SQL_ATTR_MAX_LENGTH statement attribute is intended to reduce network traffic. It is generally implemented by the data source, which truncates the data before returning it over the network. Drivers and data sources are not required to support it. Therefore, to guarantee that data is truncated to a particular size, an application should allocate a buffer of that size and specify the size in the cbValueMax argument in SQLBindCol.

3. Converts the data to the type specified by TargetType in SQLBindCol.

4. If the data was converted to a variable-length data type, such as character or binary, SQLFetch checks whether the length of the data exceeds the length of the data buffer. If the length of character data (including the null-termination character) exceeds the length of the data buffer, SQLFetch truncates the data to the length of the data buffer less the length of a null-termination character. It then null-terminates the data. If the length of binary data exceeds the length of the data buffer, SQLFetch truncates it to the length of the data buffer. The length of the data buffer is specified with BufferLength in SQLBindCol.

   SQLFetch never truncates data converted to fixed-length data types; it always assumes that the length of the data buffer is the size of the data type.

5. Puts the converted (and possibly truncated) data in the data buffer. For information about how to determine the address of the data buffer, see "Buffer Addresses" in SQLBindCol.

6. Puts the length of the data in the length/indicator buffer. If the indicator pointer and the length pointer were both set to the same buffer (as a call to SQLBindCol does), the length is written in
the buffer for valid data and SQL_NULL_DATA is written in the buffer for NULL data. If no length/indicator buffer was bound, **SQLFetch** does not return the length.

- For character or binary data, this is the length of the data after conversion and before truncation because of the data buffer being too small. If the driver cannot determine the length of the data after conversion, as is sometimes the case with long data, it sets the length to SQL_NO_TOTAL. If data was truncated because of the SQL_ATTR_MAX_LENGTH statement attribute, the value of this attribute is put in the length/indicator buffer instead of the actual length. This is because this attribute is designed to truncate data on the server before conversion, so that the driver has no way of figuring out what the actual length is.

- For all other data types, this is the length of the data after conversion; that is, it is the size of the type to which the data was converted.

For information about how to determine the address of the length/indicator buffer, see "Buffer Addresses" in SQLBindCol.

7. If the data is truncated during conversion without a loss of significant digits (for example, the real number 1.234 is truncated to the integer 1 when converted), **SQLFetch** returns SQLSTATE 01S07 (Fractional truncation) and SQL_SUCCESS_WITH_INFO. If the data is truncated because the length of the data buffer is too small (for example, the string "abcdef" is put in a 4-byte buffer), **SQLFetch** returns SQLSTATE 01004 (Data truncated) and SQL_SUCCESS_WITH_INFO. If data is truncated because of the SQL_ATTR_MAX_LENGTH statement attribute, **SQLFetch** returns SQL_SUCCESS and does not return SQLSTATE 01S07 (Fractional truncation) or SQLSTATE 01004 (Data truncated). If data is truncated during conversion with a loss of significant digits (for example, if an SQL_INTEGER value greater than 100,000 were converted to an SQL_C_TINYINT), **SQLFetch** returns SQLSTATE 22003 (Numeric value out of range) and SQL_ERROR (if the rowset size is 1) or SQL_SUCCESS_WITH_INFO (if the rowset size is greater than 1).

The contents of the bound data buffer and the length/indicator buffer are undefined if **SQLFetch** or **SQLFetchScroll** does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

**Row Status Array**

The row status array is used to return the status of each row in the rowset. The address of this array is specified with the SQL_ATTR_ROW_STATUS_PTR statement attribute. The array is allocated by the application and must have as many elements as are specified by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Its values are set by **SQLFetch**, **SQLFetchScroll**, and **SQLBulkOperations** or **SQLSetPos** (except when they have been called after the cursor has been positioned by **SQLExtendedFetch**). If the value of the SQL_ATTR_ROW_STATUS_PTR statement attribute is a null pointer, these functions do not return the row status.

The contents of the row status array buffer are undefined if **SQLFetch** or **SQLFetchScroll** does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

The following values are returned in the row status array.

<table>
<thead>
<tr>
<th>Row status array value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ROW_SUCCESS</td>
<td>The row was successfully fetched and has not changed since it</td>
</tr>
<tr>
<td>Status Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SQL_ROW_SUCCESS_WITH_INFO</td>
<td>The row was successfully fetched and has not changed since it was last fetched from this result set. However, a warning was returned about the row.</td>
</tr>
<tr>
<td>SQL_ROW_ERROR</td>
<td>An error occurred while fetching the row.</td>
</tr>
<tr>
<td>SQL_ROW_UPDATED[1],[2], and [3]</td>
<td>The row was successfully fetched and has changed since it was last fetched from this result set. If the row is fetched again from this result set or is refreshed by SQLSetPos, the status is changed to the row’s new status.</td>
</tr>
<tr>
<td>SQL_ROW_DELETED[3]</td>
<td>The row has been deleted since it was last fetched from this result set.</td>
</tr>
<tr>
<td>SQL_ROW_ADDED[4]</td>
<td>The row was inserted by SQLBulkOperations. If the row is fetched again from this result set or is refreshed by SQLSetPos, its status is SQL_ROW_SUCCESS.</td>
</tr>
<tr>
<td>SQL_ROW_NOROW</td>
<td>The rowset overlapped the end of the result set, and no row was returned that corresponded to this element of the row status array.</td>
</tr>
</tbody>
</table>

[1] For keyset, mixed, and dynamic cursors, if a key value is updated, the row of data is considered to have been deleted and a new row added.

[2] Some drivers cannot detect updates to data and therefore cannot return this value. To determine whether a driver can detect updates to refetched rows, an application calls SQLGetInfo with the SQL_ROW_UPDATES option.

[3] SQLFetch can return this value only when it is intermixed with calls to SQLFetchScroll. This is because SQLFetch moves forward through the result set and when it is used exclusively, does not refetch any rows. Because no rows are refetched, SQLFetch does not detect changes that were made to previously fetched rows. However, if SQLFetchScroll positions the cursor before any previously fetched rows and SQLFetch is used to fetch those rows, SQLFetch can detect any changes to those rows.


**Rows Fetched Buffer**

The rows fetched buffer is used to return the number of rows fetched, including those rows for which no data was returned because an error occurred while they were being fetched. In other words, it is the number of rows for which the value in the row status array is not SQL_ROW_NOROW. The address of this buffer is specified with the SQL_ATTR_ROWS_FETCHED_PTR statement attribute. The buffer is allocated by the application. It is set by SQLFetch and SQLFetchScroll. If the value of the SQL_ATTR_ROWS_FETCHED_PTR statement attribute is a null pointer, these functions do not return the number of rows fetched. To determine the number of the current row in the result set, an application can call SQLGetStmtAttr with the SQL_ATTR_ROW_NUMBER attribute.

The contents of the rows fetched buffer are undefined if SQLFetch or SQLFetchScroll does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, except when SQL_NO_DATA is returned, in
which case the value in the rows fetched buffer is set to 0.

Error Handling

Errors and warnings can apply to individual rows or to the whole function. For more information about diagnostic records, see Diagnostics and SQLGetDiagField.

Errors and Warnings on the Entire Function

If an error applies to the entire function, such as SQLSTATE HYT00 (Timeout expired) or SQLSTATE 24000 (Invalid cursor state), SQLFetch returns SQL_ERROR and the applicable SQLSTATE. The contents of the rowset buffers are undefined and the cursor position is unchanged.

If a warning applies to the entire function, SQLFetch returns SQL_SUCCESS_WITH_INFO and the applicable SQLSTATE. The status records for warnings that apply to the entire function are returned before the status records that apply to individual rows.

Errors and Warnings in Individual Rows

If an error (such as SQLSTATE 22012 (Division by zero)) or a warning (such as SQLSTATE 01004 (Data truncated)) applies to a single row, SQLFetch does the following:

- Sets the corresponding element of the row status array to SQL_ROW_ERROR for errors or SQL_ROW_SUCCESS_WITH_INFO for warnings.
- Adds zero or more status records that contain SQLSTATEs for the error or warning.
- Sets the row and column number fields in the status records. If SQLFetch cannot determine a row or column number, it sets that number to SQL_ROW_NUMBER_UNKNOWN or SQL_COLUMN_NUMBER_UNKNOWN, respectively. If the status record does not apply to a particular column, SQLFetch sets the column number to SQL_NO_COLUMN_NUMBER.

SQLFetch continues fetching rows until it has fetched all the rows in the rowset. It returns SQL_SUCCESS_WITH_INFO unless an error occurs in every row of the rowset (not including rows with status SQL_ROW_NOROW), in which case it returns SQL_ERROR. In particular, if the rowset size is 1 and an error occurs in that row, SQLFetch returns SQL_ERROR.

SQLFetch returns the status records in row number order. That is, it returns all status records for unknown rows (if any); next it returns all status records for the first row (if any), and then it returns all status records for the second row (if any), and so on. The status records for each row are ordered according to the normal rules for ordering status records; for more information, see "Sequence of Status Records" in SQLGetDiagField.

Descriptors and SQLFetch

The following sections describe how SQLFetch interacts with descriptors.
Argument Mappings

The driver does not set any descriptor fields based on the arguments of SQLFetch.

Other Descriptor Fields

The following descriptor fields are used by SQLFetch.

<table>
<thead>
<tr>
<th>Descriptor field</th>
<th>Desc.</th>
<th>Field in</th>
<th>Set through</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_ARRAY_SIZE</td>
<td>ARD</td>
<td>header</td>
<td>SQL_ATTR_ROW_ARRAY_SIZE</td>
</tr>
<tr>
<td>SQL_DESC_ARRAY_STATUS_PTR</td>
<td>IRD</td>
<td>header</td>
<td>SQL_ATTR_ROW_STATUS_PTR</td>
</tr>
<tr>
<td>SQL_DESC_BIND_OFFSET_PTR</td>
<td>ARD</td>
<td>header</td>
<td>SQL_ATTR_ROW_BIND_OFFSET_PTR</td>
</tr>
<tr>
<td>SQL_DESC_BIND_TYPE</td>
<td>ARD</td>
<td>header</td>
<td>SQL_ATTR_ROW_BIND_TYPE</td>
</tr>
<tr>
<td>SQL_DESC_COUNT</td>
<td>ARD</td>
<td>header</td>
<td>ColumnNumber argument of SQLBindCol</td>
</tr>
<tr>
<td>SQL_DESC_DATA_PTR</td>
<td>ARD</td>
<td>records</td>
<td>TargetValuePtr argument of SQLBindCol</td>
</tr>
<tr>
<td>SQL_DESC_INDICATOR_PTR</td>
<td>ARD</td>
<td>records</td>
<td>StrLen_or_IndPtr argument in SQLBindCol</td>
</tr>
<tr>
<td>SQL_DESC_OCTET_LENGTH</td>
<td>ARD</td>
<td>records</td>
<td>BufferLength argument in SQLBindCol</td>
</tr>
<tr>
<td>SQL_DESC_OCTET_LENGTH_PTR</td>
<td>ARD</td>
<td>records</td>
<td>StrLen_or_IndPtr argument in SQLBindCol</td>
</tr>
<tr>
<td>SQL_DESC_ROWS_PROCESSED_PTR</td>
<td>IRD</td>
<td>header</td>
<td>SQL_ATTR_ROWS_FETCHED_PTR</td>
</tr>
<tr>
<td>SQL_DESC_TYPE</td>
<td>ARD</td>
<td>records</td>
<td>TargetType argument in SQLBindCol</td>
</tr>
</tbody>
</table>

All descriptor fields can also be set through SQLSetDescField.
Separate Length and Indicator Buffers

Applications can bind a single buffer or two separate buffers that can be used to hold length and indicator values. When an application calls SQLBindCol, the driver sets the SQL_DESC_OCTET_LENGTH_PTR and SQL_DESC_INDICATOR_PTR fields of the ARD to the same address, which is passed in the StrLen_or_IndPtr argument. When an application calls SQLSetDescField or SQLSetDescRec, it can set these two fields to different addresses.

SQLFetch determines whether the application has specified separate length and indicator buffers. In this case, when the data is not NULL, SQLFetch sets the indicator buffer to 0 and returns the length in the length buffer. When the data is NULL, SQLFetch sets the indicator buffer to SQL_NULL_DATA and does not modify the length buffer.

Code Example

See SQLBindCol, SQLColumns, SQLGetData, and SQLProcedures.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Returning information about a column in a result set</td>
<td>SQLDescribeCol Function</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Closing the cursor on the statement</td>
<td>SQLFreeStmt Function</td>
</tr>
<tr>
<td>Fetching part or all of a column of data</td>
<td>SQLGetData Function</td>
</tr>
<tr>
<td>Returning the number of result set columns</td>
<td>SQLNumResultCols Function</td>
</tr>
<tr>
<td>Preparing a statement for execution</td>
<td>SQLPrepare Function</td>
</tr>
</tbody>
</table>

See Also
SQLFetchScroll Function

**Conformance**
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

**Summary**
SQLFetchScroll fetches the specified rowset of data from the result set and returns data for all bound columns. Rowsets can be specified at an absolute or relative position or by bookmark.

When working with an ODBC 2.x driver, the Driver Manager maps this function to SQLExtendedFetch. For more information, see Mapping Replacement Functions for Backward Compatibility of Applications.

**Syntax**

```c
SQLRETURN SQLFetchScroll(
    SQLHSTMT StatementHandle,
    SQLSMALLINT FetchOrientation,
    SQLLEN FetchOffset);
```

**Arguments**

*StatementHandle*
[Input] Statement handle.

*FetchOrientation*
[Input]
Type of fetch:
- SQL_FETCH_NEXT
- SQL_FETCH_PRIOR
- SQL_FETCH_FIRST
- SQL_FETCH_LAST
- SQL_FETCH_ABSOLUTE
- SQL_FETCH_RELATIVE
SQL_FETCH_BOOKMARK

For more information, see "Positioning the Cursor" in the "Comments" section.

FetchOffset

[Input]

Number of the row to fetch. The interpretation of this argument depends on the value of the FetchOrientation argument. For more information, see "Positioning the Cursor" in the "Comments" section.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLFetchScroll returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLFetchScroll and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise. If an error occurs on a single column, SQLGetDiagField can be called with a DiagIdentifier of SQL_DIAG_COLUMN_NUMBER to determine the column the error occurred on; and SQLGetDiagField can be called with a DiagIdentifier of SQL_DIAG_ROW_NUMBER to determine the row containing that column.

For all those SQLSTATEs that can return SQL_SUCCESS_WITH_INFO or SQL_ERROR (except 01xxx SQLSTATEs), SQL_SUCCESS_WITH_INFO is returned if an error occurs on one or more, but not all, rows of a multirow operation, and SQL_ERROR is returned if an error occurs on a single-row operation.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>String or binary data returned for a column resulted in the truncation of nonblank character or non-NULL binary data. If it was a string value, it was right-truncated.</td>
</tr>
<tr>
<td>01S01</td>
<td>Error in row</td>
<td>An error occurred while fetching one or more rows. (If this SQLSTATE is returned when an ODBC 3.x application is working with an ODBC 2.x driver, it can be ignored.)</td>
</tr>
<tr>
<td>01S06</td>
<td>Attempt to fetch before the result set</td>
<td>The requested rowset overlapped the start of the result set when FetchOrientation was SQL_FETCH_PRIOR, the current</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>returned the first rowset</td>
<td>The requested rowset overlapped the start of the result set when FetchOrientation was SQL_FETCH_PRIOR, the current position was beyond the end of the result set, and the rowset size was greater than the result set size. The requested rowset overlapped the start of the result set when FetchOrientation was SQL_FETCH_RELATIVE, FetchOffset was negative, and the absolute value of FetchOffset was less than or equal to the rowset size. The requested rowset overlapped the start of the result set when FetchOrientation was SQL_FETCH_ABSOLUTE, FetchOffset was negative, and the absolute value of FetchOffset was greater than the result set size but less than or equal to the rowset size.</td>
<td>(Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S07</td>
<td>Fractional truncation</td>
<td>The data returned for a column was truncated. For numeric data types, the fractional part of the number was truncated. For time, timestamp, and interval data types containing a time component, the fractional portion of the time was truncated. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>The data value of a column in the result set could not be converted to the data type specified by <code>TargetType</code> in <code>SQLBindCol</code>. Column 0 was bound with a data type of SQL_C_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE. Column 0 was bound with a data type of SQL_C_VARBOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was not set to SQL_UB_VARIABLE.</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>The driver was an ODBC 2.x driver that does not support <code>SQLExtendedFetch</code>, and a column number specified in the binding for a column was 0. Column 0 was bound, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>22001</td>
<td>String data, right truncated</td>
<td>A variable-length bookmark returned for a column was truncated.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Description</td>
<td>Details</td>
</tr>
<tr>
<td>------------</td>
<td>------------------</td>
<td>---------</td>
</tr>
<tr>
<td>22002</td>
<td>Indicator variable required but not supplied</td>
<td>NULL data was fetched into a column whose <code>StrLen_or_IndPtr</code> set by <code>SQLBindCol</code> (or <code>SQL_DESC_INDICATOR_PTR</code> set by <code>SQLSetDescField</code> or <code>SQLSetDescRec</code>) was a null pointer.</td>
</tr>
<tr>
<td>22003</td>
<td>Numeric value out of range</td>
<td>Returning the numeric value (as numeric or string) for one or more bound columns would have caused the whole (as opposed to fractional) part of the number to be truncated. For more information, see Converting Data from SQL to C Data Types in Appendix D: Data Types.</td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format</td>
<td>A character column in the result set was bound to a date, time, or timestamp C structure, and a value in the column was, respectively, an invalid date, time, or timestamp.</td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
<td>A value from an arithmetic expression was returned, which resulted in division by zero.</td>
</tr>
<tr>
<td>22015</td>
<td>Interval field overflow</td>
<td>Assigning from an exact numeric or interval SQL type to an interval C type caused a loss of significant digits in the leading field. When fetching data to an interval C type, there was no representation of the value of the SQL type in the interval C type.</td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast specification</td>
<td>A character column in the result set was bound to a character C buffer, and the column contained a character for which there was no representation in the character set of the buffer. The C type was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>The <code>StatementHandle</code> was in an executed state but no result set was associated with the <code>StatementHandle</code>.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction in which the fetch was executed was terminated to prevent deadlock.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <code>*MessageText</code> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
</tbody>
</table>
| HY008 | Operation canceled | Asynchronous processing was enabled for the `StatementHandle`. The function was called, and before it
completed execution, `SQLCancel` or `SQLCancelHandle` was called on the `StatementHandle`. Then the function was called again on the `StatementHandle`.

The function was called, and before it completed execution, `SQLCancel` or `SQLCancelHandle` was called on the `StatementHandle` from a different thread in a multithread application.

| HY010  | Function sequence error | (DM) An asynchronously executing function was called for the connection handle that is associated with the `StatementHandle`. This asynchronous function was still executing when the `SQLFetchScroll` function was called.

(DM) `SQLExecute`, `SQLExecDirect`, or `SQLMoreResults` was called for the `StatementHandle` and returned `SQL_PARAM_DATA_AVAILABLE`. This function was called before data was retrieved for all streamed parameters.

(DM) The specified `StatementHandle` was not in an executed state. The function was called without first calling `SQLExecDirect`, `SQLExecute` or a catalog function.

(DM) An asynchronously executing function (not this one) was called for the `StatementHandle` and was still executing when this function was called.

(DM) `SQLExecute`, `SQLExecDirect`, `SQLBulkOperations`, or `SQLSetPos` was called for the `StatementHandle` and returned `SQL_NEED_DATA`. This function was called before data was sent for all data-at-execution parameters or columns.

(DM) `SQLFetch` was called for the `StatementHandle` after `SQLExtendedFetch` was called and before `SQLFreeStmt` with the SQL_CLOSE option was called. |

| HY013  | Memory management error | The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions. |

| HY090  | Invalid string or buffer length | The SQL_ATTR_USE_BOOKMARK statement attribute was set to SQL_UB_VARIABLE, and column 0 was bound to a buffer whose length was not equal to the maximum length for the bookmark for this result set. (This length is available in the SQL_DESC_OCTET_LENGTH field of the IRD and can be obtained by calling `SQLDescribeCol`, `SQLColAttribute`, or `SQLGetDescField`.) |

| HY106  | Fetch type out of range | DM) The value specified for the argument FetchOrientation was invalid.

(DM) The argument FetchOrientation was SQL_FETCH_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.
The value of the SQL_ATTR_CURSOR_TYPE statement attribute was SQL_CURSOR_FORWARD_ONLY, and the value of argument FetchOrientation was not SQL_FETCH_NEXT.

The value of the SQL_ATTR_CURSOR_SCROLLABLE statement attribute was SQL_NONSCROLLABLE, and the value of argument FetchOrientation was not SQL_FETCH_NEXT.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY107</td>
<td>Row value out of range</td>
<td>The value specified with the SQL_ATTR_CURSOR_TYPE statement attribute was SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the SQL_ATTR_KEYSET_SIZE statement attribute was greater than 0 and less than the value specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.</td>
</tr>
<tr>
<td>HY111</td>
<td>Invalid bookmark value</td>
<td>The argument FetchOrientation was SQL_FETCH_BOOKMARK, and the bookmark pointed to by the value in the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute was not valid or was a null pointer.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The driver or data source does not support the conversion specified by the combination of the TargetType in SQLBindCol and the SQL data type of the corresponding column.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the requested result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td>SQLCompleteAsync has not been called to complete the previous asynchronous operation</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>
Comments

SQLFetchScroll returns a specified rowset from the result set. Rowsets can be specified by absolute or relative position or by bookmark. SQLFetchScroll can be called only while a result set exists — that is, after a call that creates a result set and before the cursor over that result set is closed. If any columns are bound, it returns the data in those columns. If the application has specified a pointer to a row status array or a buffer in which to return the number of rows fetched, SQLFetchScroll returns this information as well. Calls to SQLFetchScroll can be mixed with calls to SQLFetch but cannot be mixed with calls to SQLExtendedFetch.

For more information, see Using Block Cursors and Using Scrollable Cursors.

Positioning the Cursor

When the result set is created, the cursor is positioned before the start of the result set. SQLFetchScroll positions the block cursor based on the values of the FetchOrientation and FetchOffset arguments as shown in the following table. The exact rules for determining the start of the new rowset are shown in the next section.

<table>
<thead>
<tr>
<th>FetchOrientation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_FETCH_NEXT</td>
<td>Return the next rowset. This is equivalent to calling SQLFetch.</td>
</tr>
<tr>
<td></td>
<td>SQLFetchScroll ignores the value of FetchOffset.</td>
</tr>
<tr>
<td>SQL_FETCH_PRIOR</td>
<td>Return the prior rowset.</td>
</tr>
<tr>
<td></td>
<td>SQLFetchScroll ignores the value of FetchOffset.</td>
</tr>
<tr>
<td>SQL_FETCH_RELATIVE</td>
<td>Return the rowset FetchOffset from the start of the current rowset.</td>
</tr>
<tr>
<td>SQL_FETCH_ABSOLUTE</td>
<td>Return the rowset starting at row FetchOffset.</td>
</tr>
<tr>
<td>SQL_FETCH_FIRST</td>
<td>Return the first rowset in the result set.</td>
</tr>
<tr>
<td></td>
<td>SQLFetchScroll ignores the value of FetchOffset.</td>
</tr>
<tr>
<td>SQL_FETCH_LAST</td>
<td>Return the last complete rowset in the result set.</td>
</tr>
<tr>
<td></td>
<td>SQLFetchScroll ignores the value of FetchOffset.</td>
</tr>
<tr>
<td>SQL_FETCH_BOOKMARK</td>
<td>Return the rowset FetchOffset rows from the bookmark specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute.</td>
</tr>
</tbody>
</table>

Drivers are not required to support all fetch orientations; an application calls SQLGetInfo with an
information type of SQL_DYNAMIC_CURSOR_ATTRIBUTES1, SQL_KEYSET_CURSOR_ATTRIBUTES1, or SQL_STATIC_CURSOR_ATTRIBUTES1 (depending on the type of the cursor) to determine which fetch orientations are supported by the driver. The application should look at the SQL_CA1_NEXT, SQL_CA1_RELATIVE, SQL_CA1_ABSOLUTE, and WQL_CA1_BOOKMARK bitmasks in these information types. Furthermore, if the cursor is forward-only and FetchOrientation is not SQL_FETCH_NEXT, SQLFetchScroll returns SQLSTATE HY106 (Fetch type out of range).

The SQL_ATTR_ROW_ARRAY_SIZE statement attribute specifies the number of rows in the rowset. If the rowset being fetched by SQLFetchScroll overlaps the end of the result set, SQLFetchScroll returns a partial rowset. That is, if S + R – 1 is greater than L, where S is the starting row of the rowset being fetched, R is the rowset size, and L is the last row in the result set, then only the first L – S + 1 rows of the rowset are valid. The remaining rows are empty and have a status of SQL_ROW_NOROW.

After SQLFetchScroll returns, the current row is the first row of the rowset.

Cursor Positioning Rules

The following sections describe the exact rules for each value of FetchOrientation. These rules use the following notation.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before start</td>
<td>The block cursor is positioned before the start of the result set. If the first row of the new rowset is before the start of the result set, SQLFetchScroll returns SQL_NO_DATA.</td>
</tr>
<tr>
<td>After end</td>
<td>The block cursor is positioned after the end of the result set. If the first row of the new rowset is after the end of the result set, SQLFetchScroll returns SQL_NO_DATA.</td>
</tr>
<tr>
<td>CurrRowsetStart</td>
<td>The number of the first row in the current rowset.</td>
</tr>
<tr>
<td>LastResultRow</td>
<td>The number of the last row in the result set.</td>
</tr>
<tr>
<td>RowsetSize</td>
<td>The rowset size.</td>
</tr>
<tr>
<td>FetchOffset</td>
<td>The value of the FetchOffset argument.</td>
</tr>
<tr>
<td>BookmarkRow</td>
<td>The row corresponding to the bookmark specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute.</td>
</tr>
</tbody>
</table>

SQL_FETCH_NEXT

The following rules apply.

<table>
<thead>
<tr>
<th>Condition</th>
<th>First row of new rowset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition</td>
<td>First row of new rowset</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td><strong>Before start</strong></td>
<td><strong>Before start</strong></td>
</tr>
<tr>
<td><strong>CurrRowsetStart = 1</strong></td>
<td><strong>Before start</strong></td>
</tr>
<tr>
<td><strong>1 &lt;= CurrRowsetStart &lt;= RowsetSize[2]</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>CurrRowsetStart &gt; RowsetSize[2]</strong></td>
<td><strong>CurrRowsetStart - RowsetSize[2]</strong></td>
</tr>
<tr>
<td><strong>After end AND LastResultRow &lt; RowsetSize[2]</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>After end AND LastResultRow &gt;= RowsetSize[2]</strong></td>
<td><strong>LastResultRow - RowsetSize + 1[2]</strong></td>
</tr>
</tbody>
</table>

[1] **SQLFetchScroll** returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first rowset) and SQL_SUCCESS_WITH_INFO.

[2] If the rowset size has been changed since the previous call to fetch rows, this is the new rowset size.

### SQL_FETCH_PRIOR

The following rules apply.

### SQL_FETCH_RELATIVE

The following rules apply.
### SQLFetchScroll

`SQLFetchScroll` returns the same rowset as if it was called with `FetchOrientation` set to `SQL_FETCH_ABSOLUTE`. For more information, see the "SQL_FETCH_ABSOLUTE" section.

[2] `SQLFetchScroll` returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first rowset) and SQL_SUCCESS_WITH_INFO.

[3] If the rowset size has been changed since the previous call to fetch rows, this is the new rowset size.

### SQL_FETCH_ABSOLUTE

The following rules apply.

<table>
<thead>
<tr>
<th>Condition</th>
<th>First row of new rowset</th>
</tr>
</thead>
<tbody>
<tr>
<td>`FetchOffset &lt; 0 AND</td>
<td>FetchOffset</td>
</tr>
<tr>
<td>`FetchOffset &lt; 0 AND</td>
<td>FetchOffset</td>
</tr>
<tr>
<td><code>FetchOffset = 0</code></td>
<td><code>Before start</code></td>
</tr>
<tr>
<td><code>1 &lt;= FetchOffset &lt;= LastResultRow</code></td>
<td><code>FetchOffset</code></td>
</tr>
<tr>
<td><code>FetchOffset &gt; LastResultRow</code></td>
<td><code>After end</code></td>
</tr>
</tbody>
</table>
SQLFetchScroll returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first rowset) and SQL_SUCCESS_WITH_INFO.

If the rowset size has been changed since the previous call to fetch rows, this is the new rowset size.

An absolute fetch performed against a dynamic cursor cannot provide the required result because row positions in a dynamic cursor are undetermined. Such an operation is equivalent to a fetch first followed by a fetch relative; it is not an atomic operation, as is an absolute fetch on a static cursor.

**SQL_FETCH_FIRST**

The following rules apply.

<table>
<thead>
<tr>
<th>Condition</th>
<th>First row of new rowset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>1</td>
</tr>
</tbody>
</table>

**SQL_FETCH_LAST**

The following rules apply.

<table>
<thead>
<tr>
<th>Condition</th>
<th>First row of new rowset</th>
</tr>
</thead>
<tbody>
<tr>
<td>RowsetSize ([1] \leq) LastResultRow</td>
<td>LastResultRow – RowsetSize + 1([1])</td>
</tr>
<tr>
<td>RowsetSize ([1] &gt;) LastResultRow</td>
<td>1</td>
</tr>
</tbody>
</table>

\([1]\) If the rowset size has been changed since the previous call to fetch rows, this is the new rowset size.

**SQL_FETCH_BOOKMARK**

The following rules apply.

<table>
<thead>
<tr>
<th>Condition</th>
<th>First row of new rowset</th>
</tr>
</thead>
<tbody>
<tr>
<td>BookmarkRow + FetchOffset &lt; 1</td>
<td>Before start</td>
</tr>
<tr>
<td>1 (\leq) BookmarkRow + FetchOffset (\leq) LastResultRow</td>
<td>BookmarkRow + FetchOffset</td>
</tr>
<tr>
<td>BookmarkRow + FetchOffset &gt; LastResultRow</td>
<td>After end</td>
</tr>
</tbody>
</table>
For information about bookmarks, see Bookmarks (ODBC).

Effect of Deleted, Added, and Error Rows on Cursor Movement

Static and keyset-driven cursors sometimes detect rows added to the result set and remove rows deleted from the result set. By calling SQLGetInfo with the SQL_STATIC_CURSOR_ATTRIBUTES2 and SQL_KEYSET_CURSOR_ATTRIBUTES2 options and looking at the SQL_CA2_SENSITIVITY_ADDITIONS, SQL_CA2_SENSITIVITY_DELETIONS, and SQL_CA2_SENSITIVITY_UPDATES bitmasks, an application determines whether the cursors implemented by a particular driver do this. For drivers that can detect deleted rows and remove them, the following paragraphs describe the effects of this behavior. For drivers that can detect deleted rows but cannot remove them, deletions have no effect on cursor movements, and the following paragraphs do not apply.

If the cursor detects rows added to the result set or removes rows deleted from the result set, it appears as if it detects these changes only when it fetches data. This includes the case when SQLFetchScroll is called with FetchOrientation set to SQL_FETCH_RELATIVE and FetchOffset set to 0 to refetch the same rowset, but does not include the case when SQLSetPos is called with fOption set to SQL_REFRESH. In the latter case, the data in the rowset buffers is refreshed, but not refetched, and deleted rows are not removed from the result set. Thus, when a row is deleted from or inserted into the current rowset, the cursor does not modify the rowset buffers. Instead, it detects the change when it fetches any rowset that previously included the deleted row or now includes the inserted row.

For example:

```c
// Fetch the next rowset.
SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
// Delete third row of the rowset. Does not modify the rowset buffers.
SQLSetPos(hstmt, 3, SQL_DELETE, SQL_LOCK_NO_CHANGE);
// The third row has a status of SQL_ROW_DELETED after this call.
SQLSetPos(hstmt, 3, SQL_REFRESH, SQL_LOCK_NO_CHANGE);
// Refetch the same rowset. The third row is removed, replaced by what
// was previously the fourth row.
SQLFetchScroll(hstmt, SQL_FETCH_RELATIVE, 0);
```

When SQLFetchScroll returns a new rowset that has a position relative to the current rowset — that is, FetchOrientation is SQL_FETCH_NEXT, SQL_FETCH_PRIOR, or SQL_FETCH_RELATIVE — it does not include changes to the current rowset when calculating the starting position of the new rowset. However, it does include changes outside the current rowset if it is capable of detecting them. Furthermore, when SQLFetchScroll returns a new rowset that has a position independent of the current rowset — that is, FetchOrientation is SQL_FETCH_FIRST, SQL_FETCH_LAST, SQL_FETCH_ABSOLUTE, or SQL_FETCH_BOOKMARK — it includes all changes it is capable of detecting, even if they are in the current rowset.

When determining whether newly added rows are inside or outside the current rowset, a partial rowset is considered to end at the last valid row; that is, the last row for which the row status is not SQL_ROW_NOROW. For example, suppose the cursor is capable of detecting newly added rows, the current rowset is a partial rowset, the application adds new rows, and the cursor adds these rows to the end of the result set. If the application calls SQLFetchScroll with FetchOrientation set to
SQL_FETCH_NEXT, SQLFetchScroll returns the rowset starting with the first newly added row.

For example, suppose the current rowset comprises rows 21 to 30, the rowset size is 10, the cursor removes rows deleted from the result set, and the cursor detects rows added to the result set. The following table shows the rows SQLFetchScroll returns in various situations.

<table>
<thead>
<tr>
<th>Change</th>
<th>Fetch type</th>
<th>FetchOffset</th>
<th>New rowset[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete row 21</td>
<td>NEXT</td>
<td>0</td>
<td>31 to 40</td>
</tr>
<tr>
<td>Delete row 31</td>
<td>NEXT</td>
<td>0</td>
<td>32 to 41</td>
</tr>
<tr>
<td>Insert row between rows 21 and 22</td>
<td>NEXT</td>
<td>0</td>
<td>31 to 40</td>
</tr>
<tr>
<td>Insert row between rows 30 and 31</td>
<td>NEXT</td>
<td>0</td>
<td>Inserted row, 31 to 39</td>
</tr>
<tr>
<td>Delete row 21</td>
<td>PRIOR</td>
<td>0</td>
<td>11 to 20</td>
</tr>
<tr>
<td>Delete row 20</td>
<td>PRIOR</td>
<td>0</td>
<td>10 to 19</td>
</tr>
<tr>
<td>Insert row between rows 21 and 22</td>
<td>PRIOR</td>
<td>0</td>
<td>11 to 20</td>
</tr>
<tr>
<td>Insert row between rows 20 and 21</td>
<td>PRIOR</td>
<td>0</td>
<td>12 to 20, inserted row</td>
</tr>
<tr>
<td>Delete row 21</td>
<td>RELATIVE</td>
<td>0</td>
<td>22 to 31[2]</td>
</tr>
<tr>
<td>Delete row 21</td>
<td>RELATIVE</td>
<td>1</td>
<td>22 to 31</td>
</tr>
<tr>
<td>Insert row between rows 21 and 22</td>
<td>RELATIVE</td>
<td>0</td>
<td>21, inserted row, 22 to 29</td>
</tr>
<tr>
<td>Insert row between rows 21 and 22</td>
<td>RELATIVE</td>
<td>1</td>
<td>22 to 31</td>
</tr>
<tr>
<td>Delete row 21</td>
<td>ABSOLUTE</td>
<td>21</td>
<td>22 to 31[2]</td>
</tr>
<tr>
<td>Delete row 22</td>
<td>ABSOLUTE</td>
<td>21</td>
<td>21, 23 to 31</td>
</tr>
<tr>
<td>Insert row between rows 21 and 22</td>
<td>ABSOLUTE</td>
<td>22</td>
<td>Inserted row, 22 to 29</td>
</tr>
</tbody>
</table>

[1] This column uses the row numbers before any rows were inserted or deleted.

[2] In this case, the cursor attempts to return rows starting with row 21. Because row 21 has been deleted, the first row it returns is row 22.

Error rows (that is, rows with a status of SQL_ROW_ERROR) do not affect cursor movement. For example, if the current rowset starts with row 11 and the status of row 11 is SQL_ROW_ERROR, calling SQLFetchScroll with FetchOrientation set to SQL_FETCH_RELATIVE and FetchOffset set to 5 returns the rowset starting with row 16, just as it would if the status for row 11 was SQL_SUCCESS.

**Returning Data in Bound Columns**
SQLFetchScroll returns data in bound columns in the same way as SQLFetch. For more information, see "Returning Data in Bound Columns" in SQLFetch Function.

If no columns are bound, SQLFetchScroll does not return data but does move the block cursor to the specified position. Whether data can be retrieved from unbound columns of a block cursor with SQLGetData depends on the driver. This capability is supported if a call to SQLGetInfo returns the SQL_GD_BLOCK bit for the SQL_GETDATA_EXTENSIONS information type.

Buffer Addresses

SQLFetchScroll uses the same formula to determine the address of data and length/indicator buffers as SQLFetch. For more information, see "Buffer Addresses" in SQLBindCol Function.

Row Status Array

SQLFetchScroll sets values in the row status array in the same manner as SQLFetch. For more information, see "Row Status Array" in SQLFetch Function.

Rows Fetched Buffer

SQLFetchScroll returns the number of rows fetched in the rows fetched buffer in the same manner as SQLFetch. For more information, see "Rows Fetched Buffer" in SQLFetch Function.

Error Handling

When an application calls SQLFetchScroll in an ODBC 3.x driver, the Driver Manager calls SQLFetchScroll in the driver. When an application calls SQLFetchScroll in an ODBC 2.x driver, the Driver Manager calls SQLExtendedFetch in the driver. Because SQLFetchScroll and SQLExtendedFetch handle errors in a slightly different manner, the application sees slightly different error behavior when it calls SQLFetchScroll in ODBC 2.x and ODBC 3.x drivers.

SQLFetchScroll returns errors and warnings in the same manner as SQLFetch; for more information, see "Error Handling" in SQLFetch. SQLExtendedFetch returns errors in the same manner as SQLFetch, with the following exceptions:

When a warning occurs that applies to a particular row in the rowset, SQLExtendedFetch sets the corresponding entry in the row status array to SQL_ROW_SUCCESS, not SQL_ROW_SUCCESS_WITH_INFO.

If errors occur in every row in the rowset, SQLExtendedFetch returns SQL_SUCCESS_WITH_INFO, not SQL_ERROR.

In each group of status records that applies to an individual row, the first status record returned by SQLExtendedFetch must contain SQLSTATE 01S01 (Error in row); SQLFetchScroll does not return this SQLSTATE. If SQLExtendedFetch is unable to return additional SQLSTATEs, it still must return this SQLSTATE.
SQLFetchScroll and Optimistic Concurrency

If a cursor uses optimistic concurrency — that is, the SQL_ATTR_CONCURRENCY statement attribute has a value of SQL_CONCUR_VALUES or SQL_CONCUR_ROWVER — SQLFetchScroll updates the optimistic concurrency values used by the data source to detect whether a row has changed. This happens whenever SQLFetchScroll fetches a new rowset, including when it refetches the current rowset. (It is called with FetchOrientation set to SQL_FETCH_RELATIVE and FetchOffset set to 0.)

SQLFetchScroll and ODBC 2.x Drivers

When an application calls SQLFetchScroll in an ODBC 2.x driver, the Driver Manager maps this call to SQLExtendedFetch. It passes the following values for the arguments of SQLExtendedFetch.

<table>
<thead>
<tr>
<th>SQLExtendedFetch argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatementHandle</td>
<td>StatementHandle in SQLFetchScroll.</td>
</tr>
<tr>
<td>FetchOrientation</td>
<td>FetchOrientation in SQLFetchScroll.</td>
</tr>
<tr>
<td>FetchOffset</td>
<td>If FetchOrientation is not SQL_FETCH_BOOKMARK, the value of the FetchOffset argument in SQLFetchScroll is used. If FetchOrientation is SQL_FETCH_BOOKMARK, the value stored at the address specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute is used.</td>
</tr>
<tr>
<td>RowCountPtr</td>
<td>The address specified by the SQL_ATTR_ROWS_FETCHED_PTR statement attribute.</td>
</tr>
<tr>
<td>RowStatusArray</td>
<td>The address specified by the SQL_ATTR_ROW_STATUS_PTR statement attribute.</td>
</tr>
</tbody>
</table>

For more information, see Block Cursors, Scrollable Cursors, and Backward Compatibility in Appendix G: Driver Guidelines for Backward Compatibility.

Descriptors and SQLFetchScroll

SQLFetchScroll interacts with descriptors in the same manner as SQLFetch. For more information, see the "Descriptors and SQLFetchScroll" section in SQLFetch Function.

Code Example

See Column-Wise Binding, Row-Wise Binding, Positioned Update and Delete Statements, and Updating Rows in the Rowset with SQLSetPos.
### Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Performing bulk insert, update, or delete operations</td>
<td>SQLBulkOperations Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Returning information about a column in a result set</td>
<td>SQLDescribeCol Function</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Closing the cursor on the statement</td>
<td>SQLFreeStmt Function</td>
</tr>
<tr>
<td>Returning the number of result set columns</td>
<td>SQLNumResultCols Function</td>
</tr>
<tr>
<td>Positioning the cursor, refreshing data in the rowset, or updating or deleting data in the result set</td>
<td>SQLSetPos Function</td>
</tr>
<tr>
<td>Setting a statement attribute</td>
<td>SQLSetStmtAttr Function</td>
</tr>
</tbody>
</table>

### See Also

- ODBC API Reference
- ODBC Header Files

SQLForeignKeys Function

**Conformance**
Summary

`SQLForeignKeys` can return:

- A list of foreign keys in the specified table (columns in the specified table that refer to primary keys in other tables).
- A list of foreign keys in other tables that refer to the primary key in the specified table.

The driver returns each list as a result set on the specified statement.

Syntax

```c
SQLRETURN SQLForeignKeys(
    SQLHSTMT StatementHandle,
    SQLCHAR * PKCatalogName,
    SQLSMALLINT NameLength1,
    SQLCHAR * PKSchemaName,
    SQLSMALLINT NameLength2,
    SQLCHAR * PKTableName,
    SQLSMALLINT NameLength3,
    SQLCHAR * FKCatalogName,
    SQLSMALLINT NameLength4,
    SQLCHAR * FKSchemaName,
    SQLSMALLINT NameLength5,
    SQLCHAR * FKTableName,
    SQLSMALLINT NameLength6);
```

Arguments

`StatementHandle`

[Input] Statement handle.

`PKCatalogName`

[Input] Primary key table catalog name. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have catalogs. `PKCatalogName` cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, `PKCatalogName` is treated as an identifier and its case is not significant. If it is SQL_FALSE, `PKCatalogName` is an ordinary argument; it is treated literally, and its case is significant. For more information, see Arguments in Catalog Functions.

`NameLength1`

[Input] Length of `PKCatalogName`, in characters.

`PKSchemaName`

[Input] Primary key table schema name. If a driver supports schemas for some tables but not
for others, such as when the driver retrieves data from different DBMSs, an empty string (""")
denotes those tables that do not have schemas. PKSchemaName cannot contain a string search
pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, PKSchemaName is
treated as an identifier and its case is not significant. If it is SQL_FALSE, PKSchemaName is an
ordinary argument; it is treated literally, and its case is significant.

**NameLength2**

[Input] Length of *PKSchemaName, in characters.

**PKTableName**

[Input] Primary key table name. PKTableName cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, PKTableName is
treated as an identifier and its case is not significant. If it is SQL_FALSE, PKTableName is an
ordinary argument; it is treated literally, and its case is significant.

**NameLength3**

[Input] Length of *PKTableName, in characters.

**FKCatalogName**

[Input] Foreign key table catalog name. If a driver supports catalogs for some tables but not for
others, such as when the driver retrieves data from different DBMSs, an empty string (""")
denotes those tables that do not have catalogs. FKCatalogName cannot contain a string search
pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, FKCatalogName is
treated as an identifier and its case is not significant. If it is SQL_FALSE, FKCatalogName is an
ordinary argument; it is treated literally, and its case is significant.

**NameLength4**

[Input] Length of *FKCatalogName, in characters.

**FKSchemaName**

[Input] Foreign key table schema name. If a driver supports schemas for some tables but not for
others, such as when the driver retrieves data from different DBMSs, an empty string (""")
denotes those tables that do not have schemas. FKSchemaName cannot contain a string search
pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, FKSchemaName is
treated as an identifier and its case is not significant. If it is SQL_FALSE, FKSchemaName is an
ordinary argument; it is treated literally, and its case is significant.

**NameLength5**

[Input] Length of *FKSchemaName, in characters.

**FKTableName**

[Input] Foreign key table name. FKTableName cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, FKTableName is
treated as an identifier and its case is not significant. If it is SQL_FALSE, FKTableName is an
ordinary argument; it is treated literally, and its case is significant.

**NameLength6**

[Input] Length of *FKTableName, in characters.
Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLForeignKeys returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values typically returned by SQLForeignKeys and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>A cursor was open on the StatementHandle, and SQLFetch or SQLFetchScroll had been called. This error is returned by the Driver Manager if SQLFetch or SQLFetchScroll has not returned SQL_NO_DATA, and is returned by the driver if SQLFetch or SQLFetchScroll has returned SQL_NO_DATA. A cursor was open on the StatementHandle, but SQLFetch or SQLFetchScroll had not been called.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back because of a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
</tbody>
</table>
| HY008    | Operation canceled                   | Asynchronous processing was enabled for the StatementHandle. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was
The function was called, and before it completed execution, `SQLCancel` or `SQLCancelHandle` was called on the `StatementHandle` from a different thread in a multithread application.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>(DM) The arguments <code>PKTableName</code> and <code>FKTableName</code> were both null pointers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the <code>FKCatalogName</code> or <code>PKCatalogName</code> argument was a null pointer, and the SQL_CATALOG_NAME InfoType returns that catalog names are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the <code>FKSchemaName</code>, <code>PKSchemaName</code>, <code>FKTableName</code>, or <code>PKTableName</code> argument was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when the SQLForeignKeys function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for the <code>StatementHandle</code> and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) An asynchronously executing function (not this one) was called for the <code>StatementHandle</code> and was still executing when this function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for the <code>StatementHandle</code> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value of one of the name length arguments was less than 0 but not equal to SQL_NTS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value of one of the name length arguments exceeded the maximum length value for the corresponding name. (See &quot;Comments.&quot;)</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>Function Code</td>
<td>Description</td>
<td>Error Message</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>A catalog name was specified, and the driver or data source does not support catalogs. A schema name was specified, and the driver or data source does not support schemas.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes was not supported by the driver or data source. The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td>SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

**Comments**

For information about how the information returned by this function might be used, see Uses of Catalog Data.

If *PKTableName* contains a table name, SQLForeignKeys returns a result set that contains the primary key of the specified table and all the foreign keys that refer to it. The list of foreign keys in other tables does not include foreign keys that point to unique constraints in the specified table.
If *FKTableName contains a table name, SQLForeignKeys returns a result set that contains all the foreign keys in the specified table that point to primary keys in other tables, and the primary keys in the other tables to which they refer. The list of foreign keys in the specified table does not contain foreign keys that refer to unique constraints in other tables.

If both *PKTableName and *FKTableName contain table names, SQLForeignKeys returns the foreign keys in the table specified in *FKTableName that refer to the primary key of the table specified in *PKTableName. This should be one key at most.

**Note**

For more information about the general use, arguments, and returned data of ODBC catalog functions, see Catalog Functions.

SQLForeignKeys returns results as a standard result set. If the foreign keys associated with a primary key are requested, the result set is ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and KEY_SEQ. If the primary keys associated with a foreign key are requested, the result set is ordered by PKTABLE_CAT, PKTABLE_SCHEM, PKTABLE_NAME, and KEY_SEQ. The following table lists the columns in the result set.

The lengths of VARCHAR columns are not shown in the table; the actual lengths depend on the data source. To determine the actual lengths of the PKTABLE_CAT or FKTABLE_CAT, PKTABLE_SCHEM or FKTABLE_SCHEM, PKTABLE_NAME or FKTABLE_NAME, and PKCOLUMN_NAME or FKCOLUMN_NAME columns, an application can call SQLGetInfo with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKTABLE_QUALIFIER</td>
<td>PKTABLE_CAT</td>
</tr>
<tr>
<td>PKTABLE_OWNER</td>
<td>PKTABLE_SCHEM</td>
</tr>
<tr>
<td>FKTABLE_QUALIFIER</td>
<td>FK_TABLE_CAT</td>
</tr>
<tr>
<td>FKTABLE_OWNER</td>
<td>FKTABLE_SCHEM</td>
</tr>
</tbody>
</table>

The following table lists the columns in the result set. Additional columns beyond column 14 (REMARKS) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set instead of specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
</table>
| PKTABLE_CAT (ODBC 1.0) | 1 | Varchar | Primary key table catalog name; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver
retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have catalogs.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKTABLE_SCHEM</td>
<td>2</td>
<td>Varchar Primary key table schema name; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those tables that do not have schemas.</td>
</tr>
<tr>
<td>PKTABLE_NAME</td>
<td>3</td>
<td>Varchar not NULL Primary key table name.</td>
</tr>
<tr>
<td>PKCOLUMN_NAME</td>
<td>4</td>
<td>Varchar not NULL Primary key column name. The driver returns an empty string for a column that does not have a name.</td>
</tr>
<tr>
<td>FKTABLE_CAT</td>
<td>5</td>
<td>Varchar Foreign key table catalog name; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those tables that do not have catalogs.</td>
</tr>
<tr>
<td>FKTABLE_SCHEM</td>
<td>6</td>
<td>Varchar Foreign key table schema name; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those tables that do not have schemas.</td>
</tr>
<tr>
<td>FKTABLE_NAME</td>
<td>7</td>
<td>Varchar not NULL Foreign key table name.</td>
</tr>
<tr>
<td>FKCOLUMN_NAME</td>
<td>8</td>
<td>Varchar not NULL Foreign key column name. The driver returns an empty string for a column that does not have a name.</td>
</tr>
<tr>
<td>KEY_SEQ</td>
<td>9</td>
<td>Smallint not NULL Column sequence number in key (starting with 1).</td>
</tr>
<tr>
<td>UPDATE_RULE</td>
<td>10</td>
<td>Smallint Action to be applied to the foreign key when the SQL operation is UPDATE. Can have one of the following values. (The referenced table is the table that has the primary key; the referencing table is the table that has the foreign key.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL_CASCADE: When the primary key of the referenced table is updated, the foreign key of the referencing table is also updated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL_NO_ACTION: If an update of the primary key of the referenced table would cause a &quot;dangling reference&quot; in the referencing table (that is, rows in the referencing table would have no counterparts in the referenced table)</td>
</tr>
</tbody>
</table>
If an update of the foreign key of the referencing table would introduce a value that does not exist as a value of the primary key of the referenced table, the update is rejected. (This action is the same as the SQL_RESTRICT action in ODBC 2.x.)

SQL_SET_NULL: When one or more rows in the referenced table are updated in such a way that one or more components of the primary key are changed, the components of the foreign key in the referencing table that correspond to the changed components of the primary key are set to NULL in all matching rows of the referencing table.

SQL_SET_DEFAULT: When one or more rows in the referenced table are updated in such a way that one or more components of the primary key are changed, the components of the foreign key in the referencing table that correspond to the changed components of the primary key are set to the applicable default values in all matching rows of the referencing table.

NULL if not applicable to the data source.

| DELETE_RULE (ODBC 1.0) | 11 | Smallint | Action to be applied to the foreign key when the SQL operation is DELETE. Can have one of the following values. (The referenced table is the table that has the primary key; the referencing table is the table that has the foreign key.)

SQL.CASCADE: When a row in the referenced table is deleted, all the matching rows in the referencing tables are also deleted.

SQL.NO_ACTION: If a delete of a row in the referenced table would cause a "dangling reference" in the referencing table (that is, rows in the referencing table would have no counterparts in the referenced table), the update is rejected. (This action is the same as the SQL_RESTRICT action in ODBC 2.x.)

SQL_SET_NULL: When one or more rows in the referenced table are deleted, each component of the foreign key of the referencing table is set to NULL in all matching rows of the referencing table.

SQL_SET_DEFAULT: When one or more rows in the referenced table are deleted, each component of the foreign key of the referencing table is set to the applicable default in all matching rows of the referencing table.

NULL if not applicable to the data source.

| FK_NAME (ODBC 2.0) | 12 | Varchar | Foreign key name. NULL if not applicable to the data source.
PK_NAME (ODBC 2.0) 13 Varchar Primary key name. NULL if not applicable to the data source.

DEFERRABILITY (ODBC 3.0) 14 Smallint SQL_INITIALLY_DEFERRED, SQL_INITIALLY_IMMEDIATE, SQL_NOT_DEFERRABLE.

**Code Example**

As illustrated in the following table, this example uses three tables, named ORDERS, LINES, and CUSTOMERS.

<table>
<thead>
<tr>
<th>ORDERS</th>
<th>LINES</th>
<th>CUSTOMERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERID</td>
<td>ORDERID</td>
<td>CUSTID</td>
</tr>
<tr>
<td>CUSTID</td>
<td>LINES</td>
<td>NAME</td>
</tr>
<tr>
<td>OPENDATE</td>
<td>PARTID</td>
<td>ADDRESS</td>
</tr>
<tr>
<td>SALESPERSON</td>
<td>QUANTITY</td>
<td>PHONE</td>
</tr>
<tr>
<td>STATUS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the ORDERS table, CUSTID identifies the customer to whom the sale has been made. It is a foreign key that refers to CUSTID in the CUSTOMERS table.

In the LINES table, ORDERID identifies the sales order with which the line item is associated. It is a foreign key that refers to ORDERID in the ORDERS table.

This example calls SQLPrimaryKeys to get the primary key of the ORDERS table. The result set will have one row; the significant columns are shown in the following table.

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>COLUMN_NAME</th>
<th>KEY_SEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERS</td>
<td>ORDERID</td>
<td>1</td>
</tr>
</tbody>
</table>

Next, the example calls SQLForeignKeys to get the foreign keys in other tables that reference the primary key of the ORDERS table. The result set will have one row; the significant columns are shown in the following table.

<table>
<thead>
<tr>
<th>PKTABLE_NAME</th>
<th>PKCOLUMN_NAME</th>
<th>FKTABLE_NAME</th>
<th>FKCOLUMN_NAME</th>
<th>KEY_SEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERS</td>
<td>CUSTID</td>
<td>LINES</td>
<td>CUSTID</td>
<td>1</td>
</tr>
</tbody>
</table>

Finally, the example calls SQLForeignKeys to get the foreign keys in the ORDERS table that refer to
the primary keys of other tables. The result set will have one row; the significant columns are shown in the following table.

<table>
<thead>
<tr>
<th>PKTABLE_NAME</th>
<th>PKCOLUMN_NAME</th>
<th>FKTABLE_NAME</th>
<th>FKCOLUMN_NAME</th>
<th>KEY_SEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMERS</td>
<td>CUSTID</td>
<td>ORDERS</td>
<td>CUSTID</td>
<td>1</td>
</tr>
</tbody>
</table>

```c
#define TAB_LEN SQL_MAX_TABLE_NAME_LEN + 1
#define COL_LEN SQL_MAX_COLUMN_NAME_LEN + 1

LPSTR szTable;    /* Table to display */
UCHAR szPkTable[TAB_LEN];    /* Primary key table name */
UCHAR szFkTable[TAB_LEN];    /* Foreign key table name */
UCHAR szPkCol[COL_LEN];      /* Primary key column */
UCHAR szFkCol[COL_LEN];      /* Foreign key column */

SQLHSTMT hstmt;
SQLINTEGER cbPkTable, cbPkCol, cbFkTable, cbFkCol, cbKeySeq;
SQLSMALLINT iKeySeq;
SQLRETURN retcode;

// Bind the columns that describe the primary and foreign keys.
// Ignore the table schema, name, and catalog for this example.

SQLBindCol(hstmt, 3, SQL_C_CHAR, szPkTable, TAB_LEN, &cbPkTable);
SQLBindCol(hstmt, 4, SQL_C_CHAR, szPkCol, COL_LEN, &cbPkCol);
SQLBindCol(hstmt, 5, SQL_C_SSHORT, &iKeySeq, TAB_LEN, &cbKeySeq);
SQLBindCol(hstmt, 7, SQL_C_CHAR, szFkTable, TAB_LEN, &cbFkTable);
SQLBindCol(hstmt, 8, SQL_C_CHAR, szFkCol, COL_LEN, &cbFkCol);

strcpy_s(szTable, sizeof(szTable), "ORDERS");

/* Get the names of the columns in the primary key. */

retcode = SQLPrimaryKeys(hstmt,
    NULL, 0,    /* Catalog name */
    NULL, 0,    /* Schema name */
    szTable, SQL_NTS);    /* Table name */

while ((retcode == SQL_SUCCESS) || (retcode == SQL_SUCCESS_WITH_INFO)) {
    /* Fetch and display the result set. This will be a list of the */
    /* columns in the primary key of the ORDERS table. */

    retcode = SQLFetch(hstmt);
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
        fprintf(out, "Table: %s Column: %s Key Seq: %hd \n", szPkTable, szPkCol, iKeySeq);
}

/* Close the cursor (the hstmt is still allocated). */
SQLFreeStmt(hstmt, SQL_CLOSE);
```
/* Get all the foreign keys that refer to ORDERS primary key. */

retcode = SQLForeignKeys(hstmt,
    NULL, 0, /* Primary catalog */
    NULL, 0, /* Primary schema */
    szTable, SQL_NTS, /* Primary table */
    NULL, 0, /* Foreign catalog */
    NULL, 0, /* Foreign schema */
    NULL, 0); /* Foreign table */

while ((retcode == SQL_SUCCESS) || (retcode == SQL_SUCCESS_WITH_INFO)) {

    /* Fetch and display the result set. This will be all of the */
    /* foreign keys in other tables that refer to the ORDERS */
    /* primary key. */

    retcode = SQLFetch(hstmt);
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
        fprintf(out, "%s (%s) <-- %s (%s)\n", szPkTable,
            szPkCol, szFkTable, szFkCol);
}

/* Close the cursor (the hstmt is still allocated). */
SQLFreeStmt(hstmt, SQL_CLOSE);

/* Get all the foreign keys in the ORDERS table. */

retcode = SQLForeignKeys(hstmt,
    NULL, 0, /* Primary catalog */
    NULL, 0, /* Primary schema */
    NULL, 0, /* Primary table */
    NULL, 0, /* Foreign catalog */
    NULL, 0, /* Foreign schema */
    szTable, SQL_NTS); /* Foreign table */

while ((retcode == SQL_SUCCESS) || (retcode == SQL_SUCCESS_WITH_INFO)) {

    /* Fetch and display the result set. This will be all of the */
    /* primary keys in other tables that are referred to by foreign */
    /* keys in the ORDERS table. */

    retcode = SQLFetch(hstmt);
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
        fprintf(out, "%s (%s) --> %s (%s )\n", szFkTable, szFkCol,
            szPkTable, szPkCol);
}

/* Free the hstmt. */
SQLFreeStmt(hstmt, SQL_DROP);

Related Functions
Binding a buffer to a column in a result set  |  SQLBindCol Function
---|---
Canceling statement processing  |  SQLCancel Function
Fetching a single row or a block of data in a forward-only direction  |  SQLFetch Function
Fetching a block of data or scrolling through a result set  |  SQLFetchScroll Function
Returning the columns of a primary key  |  SQLPrimaryKeys Function
Returning table statistics and indexes  |  SQLStatistics Function

See Also
ODBC API Reference
ODBC Header Files

SQLFreeConnect Function

**Conformance**
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

**Summary**
In ODBC 3.x, the ODBC 2.0 function `SQLFreeConnect` has been replaced by `SQLFreeHandle`. For more information, see `SQLFreeHandle`.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
</table>
For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.

See Also
ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLFreeEnv Function
**Conformance**
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

**Summary**
In ODBC 3.x, the ODBC 2.0 function `SQLFreeEnv` has been replaced by `SQLFreeHandle`. For more information, see `SQLFreeHandle`.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see <code>Mapping Deprecated Functions</code> in Appendix G: Driver Guidelines for Backward Compatibility.</td>
</tr>
</tbody>
</table>

**See Also**
- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft

---

**SQLFreeHandle Function**

**Conformance**
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

**Summary**
`SQLFreeHandle` frees resources associated with a specific environment, connection, statement, or descriptor handle.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This function is a generic function for freeing handles. It replaces the ODBC 2.0 functions <code>SQLFreeConnect</code> (for freeing a connection handle) and <code>SQLFreeEnv</code> (for freeing an environment handle). <code>SQLFreeConnect</code> and <code>SQLFreeEnv</code> are both deprecated in ODBC 3.x. <code>SQLFreeHandle</code> also replaces the ODBC 2.0 function <code>SQLFreeStmt</code> (with the SQL_DROP Option) for freeing a statement handle. For more information, see &quot;Comments.&quot; For more information about what the Driver Manager maps this function to when an ODBC 3.x application is working with an ODBC 2.x driver, see <code>Mapping Replacement Functions for Backward Compatibility of Applications</code>.</td>
</tr>
</tbody>
</table>

**Syntax**

```c
SQLRETURN SQLFreeHandle(
    SQLSMALLINT HandleType,
    SQLHANDLE Handle);
```
Arguments

(HandleType) [Input] The type of handle to be freed by SQLFreeHandle. Must be one of the following values:

- SQL_HANDLE_DBC
- SQL_HANDLE_DBC_INFO_TOKEN
- SQL_HANDLE_DESC
- SQL_HANDLE_ENV
- SQL_HANDLE_STMT

SQL_HANDLE_DBC_INFO_TOKEN handle is used only by the Driver Manager and driver. Applications should not use this handle type. For more information about SQL_HANDLE_DBC_INFO_TOKEN, see Developing Connection-Pool Awareness in an ODBC Driver.

If HandleType is not one of these values, SQLFreeHandle returns SQL_INVALID_HANDLE.

(Handle) [Input] The handle to be freed.

Returns

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE.

If SQLFreeHandle returns SQL_ERROR, the handle is still valid.

Diagnostics

When SQLFreeHandle returns SQL_ERROR, an associated SQLSTATE value may be obtained from the diagnostic data structure for the handle that SQLFreeHandle tried to free but could not. The following table lists the SQLSTATE values typically returned by SQLFreeHandle and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in</td>
</tr>
</tbody>
</table>
The *MessageText buffer describes the error and its cause.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) The HandleType argument was SQL_HANDLE_ENV, and at least one connection was in an allocated or connected state. SQLDisconnect and SQLFreeHandle with a HandleType of SQL_HANDLE_DBC must be called for each connection before calling SQLFreeHandle with a HandleType of SQL_HANDLE_ENV. (DM) The HandleType argument was SQL_HANDLE_DBC, and the function was called before calling SQLDisconnect for the connection. (DM) The HandleType argument was SQL_HANDLE_DBC. An asynchronously executing function was called with Handle and the function was still executing when this function was called. (DM) The HandleType argument was SQL_HANDLE_STMT. SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called with the statement handle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) The HandleType argument was SQL_HANDLE_STMT. An asynchronously executing function was called on the statement handle or on the associated connection handle and the function was still executing when this function was called. (DM) The HandleType argument was SQL_HANDLE_DESC. An asynchronously executing function was called on the associated connection handle; and the function was still executing when this function was called. (DM) All subsidiary handles and other resources were not released before SQLFreeHandle was called. (DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for one of the statement handles associated with the Handle and HandleType was set to SQL_HANDLE_STMT or SQL_HANDLE_DESC returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The HandleType argument was SQL_HANDLE_STMT or SQL_HANDLE_DESC, and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY017</td>
<td>Invalid use of an automatically allocated descriptor handle.</td>
<td>(DM) The Handle argument was set to the handle for an automatically allocated descriptor.</td>
</tr>
</tbody>
</table>
### HY117
Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.

(DM) For more information about suspended state, see `SQLEndTran` Function.

### HYT01
Connection timeout expired

The connection timeout period expired before the data source responded to the request. The connection timeout period is set through `SQLSetConnectAttr`, `SQL_ATTR_CONNECTION_TIMEOUT`.

### IM001
Driver does not support this function

(DM) The `HandleType` argument was SQL_HANDLE_DESC, and the driver was an ODBC 2.x driver.

(DM) The `HandleType` argument was SQL_HANDLE_STMT, and the driver was not a valid ODBC driver.

### Comments

`SQLFreeHandle` is used to free handles for environments, connections, statements, and descriptors, as described in the following sections. For general information about handles, see Handles.

An application should not use a handle after it has been freed; the Driver Manager does not check the validity of a handle in a function call.

#### Freeing an Environment Handle

Before it calls `SQLFreeHandle` with a `HandleType` of SQL_HANDLE_ENV, an application must call `SQLFreeHandle` with a `HandleType` of SQL_HANDLE_DBC for all connections allocated under the environment. Otherwise, the call to `SQLFreeHandle` returns SQL_ERROR and the environment and any active connection remains valid. For more information, see Environment Handles and Allocating the Environment Handle.

If the environment is a shared environment, the application that calls `SQLFreeHandle` with a `HandleType` of SQL_HANDLE_ENV no longer has access to the environment after the call, but the environment’s resources are not necessarily freed. The call to `SQLFreeHandle` decrements the reference count of the environment. The reference count is maintained by the Driver Manager. If it does not reach zero, the shared environment is not freed, because it is still being used by another component. If the reference count reaches zero, the resources of the shared environment are freed.

#### Freeing a Connection Handle

Before it calls `SQLFreeHandle` with a `HandleType` of SQL_HANDLE_DBC, an application must call `SQLDisconnect` for the connection if there is a connection on this handle. Otherwise, the call to `SQLFreeHandle` returns SQL_ERROR and the connection remains valid.
Freeing a Statement Handle

A call to SQLFreeHandle with a HandleType of SQL_HANDLE_STMT frees all resources that were allocated by a call to SQLAllocHandle with a HandleType of SQL_HANDLE_STMT. When an application calls SQLFreeHandle to free a statement that has pending results, the pending results are deleted. When an application frees a statement handle, the driver frees the four automatically allocated descriptors associated with that handle. For more information, see Statement Handles and Freeing a Statement Handle.

Notice that SQLDisconnect automatically drops any statements and descriptors open on the connection.

Freeing a Descriptor Handle

A call to SQLFreeHandle with a HandleType of SQL_HANDLE_DESC frees the descriptor handle in Handle. The call to SQLFreeHandle does not release any memory allocated by the application that may be referenced by a pointer field (including SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR) of any descriptor record of Handle. The memory allocated by the driver for fields that are not pointer fields is freed when the handle is freed. When a user-allocated descriptor handle is freed, all statements that the freed handle had been associated with revert to their respective automatically allocated descriptor handles.

\[\text{Note}\]

ODBC 2.x drivers do not support freeing descriptor handles, just as they do not support allocating descriptor handles.

Notice that SQLDisconnect automatically drops any statements and descriptors open on the connection. When an application frees a statement handle, the driver frees all the automatically generated descriptors associated with that handle.

For more information about descriptors, see Descriptors.

Code Example

For additional code samples, see SQLBrowseConnect and SQLConnect.

Code

```c
// SQLFreeHandle.cpp
// compile with: user32.lib odbc32.lib
#include <windows.h>
#include <sqlext.h>
#include <stdio.h>
```
```c
int main()
{
    SQLRETURN retCode;
    HWND desktopHandle = GetDesktopWindow(); // desktop's window handle
    SQLCHAR connStrbuffer[1024];
    SQLSMALLINT connStrBufferLen;

    // Initialize the environment, connection, statement handles.
    SQLHENV henv = NULL; // Environment
    SQLHDBC hdbc = NULL; // Connection handle
    SQLHSTMT hstmt = NULL; // Statement handle

    // Allocate the environment.
    retCode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

    // Set environment attributes.
    retCode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, -1);

    // Allocate the connection.
    retCode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

    // Set the login timeout.
    retCode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)10, 0);

    // Let the user select the data source and connect to the database.
    retCode = SQLDriverConnect(hdbc, desktopHandle, (SQLCHAR *)"Driver={SQL Server}", SQL_NTS, connStrbuffer, 1025, &connStrBufferLen, SQL_DRIVER_PROMPT);

    retCode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

    // Free handles, and disconnect.
    if (hstmt) {
        SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
        hstmt = NULL;
    }
    if (hdbc) {
        SQLDisconnect(hdbc);
        SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
        hdbc = NULL;
    }
    if (henv) {
        SQLFreeHandle(SQL_HANDLE_ENV, henv);
        henv = NULL;
    }
}
```

### Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocating a handle</td>
<td>SQLAllocHandle Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
</tbody>
</table>
SQLFreeStmt Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLFreeStmt stops processing associated with a specific statement, closes any open cursors associated with the statement, discards pending results, or, optionally, frees all resources associated with the statement handle.

Syntax

```c
SQLRETURN SQLFreeStmt(
    SQLHSTMT StatementHandle,
    SQLUSMALLINT Option);
```

Arguments

*StatementHandle*
[Input] Statement handle

*Option*
[Input] One of the following options:

SQL_CLOSE: Closes the cursor associated with *StatementHandle* (if one was defined) and discards all pending results. The application can reopen this cursor later by executing a **SELECT** statement again with the same or different parameter values. If no cursor is open, this option has no effect for the application. **SQLCloseCursor** can also be called to close a cursor. For more information, see Closing the Cursor.

SQL_DROP: This option is deprecated. A call to **SQLFreeStmt** with an *Option* of SQL_DROP is mapped in the Driver Manager to **SQLFreeHandle**.
SQL_UNBIND: Sets the SQL_DESC_COUNT field of the ARD to 0, releasing all column buffers bound by SQLBindCol for the given StatementHandle. This does not unbind the bookmark column; to do that, the SQL_DESC_DATA_PTR field of the ARD for the bookmark column is set to NULL. Notice that if this operation is performed on an explicitly allocated descriptor that is shared by more than one statement, the operation will affect the bindings of all statements that share the descriptor. For more information, see Overview of Retrieving Results (Basic).

SQL_RESET_PARAMS: Sets the SQL_DESC_COUNT field of the APD to 0, releasing all parameter buffers set by SQLBindParameter for the given StatementHandle. If this operation is performed on an explicitly allocated descriptor that is shared by more than one statement, this operation will affect the bindings of all the statements that share the descriptor. For more information, see Binding Parameters.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When SQLFreeStmt returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values typically returned by SQLFreeStmt and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when SQLFreeStmt was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called with Option set to SQL_RESET_PARAMS before data was retrieved for all streamed columns.</td>
</tr>
</tbody>
</table>
parameters.

(DM) An asynchronously executing function was called for the StatementHandle and was still executing when this function was called.

(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
</tr>
<tr>
<td>HY092</td>
<td>Option type out of range</td>
</tr>
<tr>
<td>HY102</td>
<td>Connection timeout expired</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
</tr>
</tbody>
</table>

Comments

Calling SQLFreeStmt with the SQL_CLOSE option is equivalent to calling SQLCloseCursor, except that SQLFreeStmt with SQL_CLOSE does not affect the application if no cursor is open on the statement. If no cursor is open, a call to SQLCloseCursor returns SQLSTATE 24000 (Invalid cursor state).

An application should not use a statement handle after it has been freed; the Driver Manager does not check the validity of a handle in a function call.

It is a good programming practice to free handles. However, for simplicity, the following sample does not include code that frees allocated handles. For an example of how to free handles, see SQLFreeHandle Function.

```c
#include <windows.h>
#include <sqlext.h>

int main() {
    // declare and initialize the environment, connection, statement handles
    SQLHENV henv = NULL;  // Environment
    SQLHDBC hdbc = NULL;  // Connection handle
```
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocating a handle</td>
<td>SQLAllocHandle Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Closing a cursor</td>
<td>SQLCloseCursor Function</td>
</tr>
<tr>
<td>Freeing a handle</td>
<td>SQLFreeHandle Function</td>
</tr>
<tr>
<td>Setting a cursor name</td>
<td>SQLSetCursorName Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

SQLGetConnectAttr Function

Conformance
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary
SQLGetConnectAttr returns the current setting of a connection attribute.
Note

For more information about what the Driver Manager maps this function to when an ODBC 3.x application is working with an ODBC 2.x driver, see Mapping Replacement Functions for Backward Compatibility of Applications.

Syntax

```c
SQLRETURN SQLGetConnectAttr(
    SQLHDBC ConnectionHandle,
    SQLINTEGER Attribute,
    SQLPOINTER ValuePtr,
    SQLINTEGER BufferLength,
    SQLINTEGER * StringLengthPtr);
```

Arguments

**ConnectionHandle**

[Input] Connection handle.

**Attribute**

[Input] Attribute to retrieve.

**ValuePtr**

[Output] A pointer to memory in which to return the current value of the attribute specified by **Attribute**. For integer-type attributes, some drivers may only write the lower 32-bit or 16-bit of a buffer and leave the higher-order bit unchanged. Therefore, applications should use a buffer of SQLULEN and initialize the value to 0 before calling this function.

If **ValuePtr** is NULL, **StringLengthPtr** will still return the total number of bytes (excluding the null-termination character for character data) available to return in the buffer pointed to by **ValuePtr**.

**BufferLength**

[Input] If **Attribute** is an ODBC-defined attribute and **ValuePtr** points to a character string or a binary buffer, this argument should be the length of **ValuePtr**. If **Attribute** is an ODBC-defined attribute and **ValuePtr** is an integer, **BufferLength** is ignored. If the value in **ValuePtr** is a Unicode string (when calling SQLGetConnectAttrW), the **BufferLength** argument must be an even number.

If **Attribute** is a driver-defined attribute, the application indicates the nature of the attribute to the Driver Manager by setting the **BufferLength** argument. **BufferLength** can have the following values:

- If **ValuePtr** is a pointer to a character string, **BufferLength** is the length of the string.
- If **ValuePtr** is a pointer to a binary buffer, the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in **BufferLength**. This places a negative value in **BufferLength**.
• If *ValuePtr is a pointer to a value other than a character string or binary string, BufferLength should have the value SQL_IS_POINTER.

• If *ValuePtr contains a fixed-length data type, BufferLength is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.

**StringLengthPtr**

[Output] A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character) available to return in *ValuePtr. If *ValuePtr is a null pointer, no length is returned. If the attribute value is a character string and the number of bytes available to return is greater than BufferLength minus the length of the null-termination character, the data in *ValuePtr is truncated to BufferLength minus the length of the null-termination character and is null-terminated by the driver.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When SQLGetConnectAttr returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained from the diagnostic data structure by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_DBC and a Handle of ConnectionHandle. The following table lists the SQLSTATE values typically returned by SQLGetConnectAttr and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
</tr>
</tbody>
</table>
| HY000    | General error     | An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
</tbody>
</table>
| HY010 | Function sequence error                          | (DM) **SQLBrowseConnect** was called for the `ConnectionHandle` and returned SQL_NEED_DATA. This function was called before **SQLBrowseConnect** returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.  
  (DM) **SQLExecute, SQLExecDirect, or SQLMoreResults** was called for the `ConnectionHandle` and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. |
| HY013 | Memory management error                          | The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.                                                         |
| HY090 | Invalid string or buffer length                  | (DM) *ValuePtr* is a character string, and BufferLength was less than zero but not equal to SQL_NTS.                                                                                                   |
| HY092 | Invalid attributeoption identifier               | The value specified for the argument `Attribute` was not valid for the version of ODBC supported by the driver.                                                                                         |
| HY114 | Driver does not support connection-level asynchronous function execution | (DM) An application attempted to enable asynchronous function execution with SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE for a driver that does not support asynchronous connection operations. |
| HY117 | Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed. | (DM) For more information about suspended state, see **SQLEndTran Function**.                                                                                                                         |
| HYC00 | Optional feature not implemented                 | The value specified for the argument `Attribute` was a valid ODBC connection attribute for the version of ODBC supported by the driver, but was not supported by the driver. |
| HYT01 | Connection timeout expired                       | The connection timeout period expired before the data source responded to the request. The connection timeout period is set through **SQLSetConnectAttr**, SQL_ATTR_CONNECTION_TIMEOUT. |
| IM001 | Driver does not support this function            | (DM) The driver that corresponds to the `ConnectionHandle` does not support the function.                                                                                                           |
Comments

For general information about connection attributes, see Connection Attributes.

For a list of attributes that can be set, see SQLSetConnectAttr. Notice that if Attribute specifies an attribute that returns a string, ValuePtr must be a pointer to a buffer for the string. The maximum length of the returned string, including the null-termination character, will be BufferLength bytes.

Depending on the attribute, an application does not have to establish a connection before calling SQLGetConnectAttr. However, if SQLGetConnectAttr is called and the specified attribute does not have a default and has not been set by a prior call to SQLSetConnectAttr, SQLGetConnectAttr will return SQL_NO_DATA.

If Attribute is SQL_ATTR_TRACE or SQL_ATTR_TRACEFILE, ConnectionHandle does not have to be valid, and SQLGetConnectAttr will not return SQL_ERROR or SQL_INVALID_HANDLE if ConnectionHandle is invalid. These attributes apply to all connections. SQLGetConnectAttr will return SQL_ERROR or SQL_INVALID_HANDLE if another argument is invalid.

Although an application can set statement attributes by using SQLSetConnectAttr, an application cannot use SQLGetConnectAttr to retrieve statement attribute values; it must call SQLGetStmtAttr to retrieve the setting of statement attributes.

Both SQL_ATTR_AUTO_IPD and SQL_ATTR_CONNECTION_DEAD connection attributes can be returned by a call to SQLGetConnectAttr but cannot be set by a call to SQLSetConnectAttr.

Note

There is no asynchronous support for SQLGetConnectAttr. When implementing SQLGetConnectAttr, a driver can improve performance by minimizing the number of times that information is sent or requested from the server.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returning the setting of a statement attribute</td>
<td>SQLGetStmtAttr Function</td>
</tr>
<tr>
<td>Setting a connection attribute</td>
<td>SQLSetConnectAttr Function</td>
</tr>
<tr>
<td>Setting an environment attribute</td>
<td>SQLSetEnvAttr Function</td>
</tr>
<tr>
<td>Setting a statement attribute</td>
<td>SQLSetStmtAttr Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files
SQLGetConnectOption Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

Summary
In ODBC 3.x, the ODBC 2.x function SQLGetConnectOption has been replaced by SQLGetConnectAttr. For more information, see SQLGetConnectAttr.

Note
For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.

Note
The attribute SQL_ASYNC_DBC_FUNCTION_ENABLE introduced in ODBC 3.8 is not supported by SQLGetConnectOption. Applications that use the asynchronous operation on a connection handle must use SQLGetConnectAttr.

See Also
ODBC API Reference
ODBC Header Files

SQLGetCursorName Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLGetCursorName returns the cursor name associated with a specified statement.

Syntax

```
SQLRETURN SQLGetCursorName(
    SQLHSTMT StatementHandle,
```
Arguments

StatementHandle
[Input] Statement handle.

CursorName
[Output] Pointer to a buffer in which to return the cursor name.

If CursorName is NULL, NameLengthPtr will still return the total number of characters (excluding the null-termination character for character data) available to return in the buffer pointed to by CursorName.

BufferLength
[Input] Length of *CursorName, in characters. If the value in *CursorName is a Unicode string (when calling SQLGetCursorNameW), the BufferLength argument must be an even number.

NameLengthPtr
[Output] Pointer to memory in which to return the total number of characters (excluding the null-termination character) available to return in *CursorName. If the number of characters available to return is greater than or equal to BufferLength, the cursor name in *CursorName is truncated to BufferLength minus the length of a null-termination character.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLGetCursorName returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLGetCursorName and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The buffer *CursorName was not large enough to return the entire cursor name, so the cursor name was truncated. The length of the untruncated cursor name is</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLGetCursorName function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) An asynchronously executing function was called for the StatementHandle and was still executing when this function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY015</td>
<td>No cursor name available</td>
<td>(DM) The driver was an ODBC 2.x driver, there was no open cursor on the statement, and no cursor name had been set with SQLSetCursorName.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value specified in the argument BufferLength was less than 0.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr,</td>
</tr>
</tbody>
</table>
Comments

Cursor names are used only in positioned update and delete statements (for example, `UPDATE table-name ...WHERE CURRENT OF cursor-name`). For more information, see Positioned Update and Delete Statements. If the application does not call `SQLSetCursorName` to define a cursor name, the driver generates a name. This name begins with the letters `SQL_CUR`.

**Note**

In ODBC 2.x, when there was no open cursor and no name had been set by a call to `SQLSetCursorName`, a call to `SQLGetCursorName` returned SQLSTATE HY015 (No cursor name available). In ODBC 3.x, this is no longer true; regardless of when `SQLGetCursorName` is called, the driver returns the cursor name.

`SQLGetCursorName` returns the name of a cursor whether or not the name was created explicitly or implicitly. A cursor name is implicitly generated if `SQLSetCursorName` is not called. `SQLSetCursorName` can be called to rename a cursor on a statement as long as the cursor is in an allocated or prepared state.

A cursor name that is set either explicitly or implicitly remains set until the `StatementHandle` with which it is associated is dropped, using `SQLFreeHandle` with a `HandleType` of SQL_HANDLE_STMT.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executing an SQL statement</td>
<td><code>SQLExecDirect Function</code></td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td><code>SQLExecute Function</code></td>
</tr>
<tr>
<td>Preparing a statement for execution</td>
<td><code>SQLPrepare Function</code></td>
</tr>
<tr>
<td>Setting a cursor name</td>
<td><code>SQLSetCursorName Function</code></td>
</tr>
</tbody>
</table>

See Also

- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft
SQLGetData Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLGetData retrieves data for a single column in the result set or for a single parameter after SQLParamData returns SQL_PARAM_DATA_AVAILABLE. It can be called multiple times to retrieve variable-length data in parts.

Syntax

```
SQLRETURN SQLGetData(
    SQLHSTMT StatementHandle,
    SQLUSMALLINT Col_or_Param_Num,
    SQLSMALLINT TargetType,
    SQLPOINTER TargetValuePtr,
    SQLLEN BufferLength,
    SQLLEN * StrLen_or_IndPtr);
```

Arguments

**StatementHandle**
[Input] Statement handle.

**Col_or_Param_Num**
[Input] For retrieving column data, it is the number of the column for which to return data. Result set columns are numbered in increasing column order starting at 1. The bookmark column is column number 0; this can be specified only if bookmarks are enabled.

For retrieving parameter data, it is the ordinal of the parameter, which starts at 1.

**TargetType**
[Input] The type identifier of the C data type of the *TargetValuePtr buffer. For a list of valid C data types and type identifiers, see the C Data Types section in Appendix D: Data Types.

If TargetType is SQL_ARD_TYPE, the driver uses the type identifier specified in the SQL_DESC_CONCISE_TYPE field of the ARD. If TargetType is SQL_APD_TYPE, SQLGetData will use the same C data type that was specified in SQLBindParameter. Otherwise, the C data type specified in SQLGetData overrides the C data type specified in SQLBindParameter. If it is SQL_C_DEFAULT, the driver selects the default C data type based on the SQL data type of the source.

You can also specify an extended C data type. For more information, see C Data Types in ODBC.

**TargetValuePtr**
[Output] Pointer to the buffer in which to return the data.
TargetValuePtr cannot be NULL.

**BufferLength**

[Input] Length of the *TargetValuePtr buffer in bytes.

The driver uses BufferLength to avoid writing past the end of the *TargetValuePtr buffer when returning variable-length data, such as character or binary data. Note that the driver counts the null-termination character when returning character data to *TargetValuePtr. *TargetValuePtr must therefore contain space for the null-termination character, or the driver will truncate the data.

When the driver returns fixed-length data, such as an integer or a date structure, the driver ignores BufferLength and assumes the buffer is large enough to hold the data. It is therefore important for the application to allocate a large enough buffer for fixed-length data or the driver will write past the end of the buffer.

**SQLGetData** returns SQLSTATE HY090 (Invalid string or buffer length) when BufferLength is less than 0 but not when BufferLength is 0.

**StrLen_or_IndPtr**

[Output] Pointer to the buffer in which to return the length or indicator value. If this is a null pointer, no length or indicator value is returned. This returns an error when the data being fetched is NULL.

**SQLGetData** can return the following values in the length/indicator buffer:

- The length of the data available to return
- SQL_NO_TOTAL
- SQL_NULL_DATA

For more information, see Using Length/Indicator Values and "Comments" in this topic.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When **SQLGetData** returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling **SQLGetDiagRec** with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by **SQLGetData** and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Message</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>Not all of the data for the specified column, <code>Col_or_Param_Num</code>, could be retrieved in a single call to the function. SQL_NO_TOTAL or the length of the data remaining in the specified column prior to the current call to <code>SQLGetData</code> is returned in *StrLen_or_IndPtr. (Function returns SQL_SUCCESS_WITH_INFO.) For more information on using multiple calls to <code>SQLGetData</code> for a single column, see &quot;Comments.&quot;</td>
</tr>
<tr>
<td>01S07</td>
<td>Fractional truncation</td>
<td>The data returned for one or more columns was truncated. For numeric data types, the fractional part of the number was truncated. For time, timestamp, and interval data types containing a time component, the fractional portion of the time was truncated. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>The data value of a column in the result set cannot be converted to the C data type specified by the argument <code>TargetType</code>.</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>The value specified for the argument <code>Col_or_Param_Num</code> was 0, and the <code>SQL_ATTR_USE_BOOKMARKS</code> statement attribute was set to SQL_UB_OFF.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value specified for the argument <code>Col_or_Param_Num</code> was greater than the number of columns in the result set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>Col_or_Param_Num</code> value was not equal to the ordinal of the parameter that is available.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The specified column was bound. This description does not apply to drivers that return the SQL_GD_BOUND bitmask for the SQL_GETDATA_EXTENSIONS option in <code>SQLGetData</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The number of the specified column was less than or equal to the number of the highest bound column. This description does not apply to drivers that return the SQL_GD_ANY_COLUMN bitmask for the SQL_GETDATA_EXTENSIONS option in <code>SQLGetData</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The application has already called <code>SQLGetData</code> for the current row; the number of the column specified in the current call was less than the number of the column specified in the preceding call; and the driver does not return the SQL_GD_ANY_ORDER bitmask for the SQL_GETDATA_EXTENSIONS option in <code>SQLGetData</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The <code>TargetType</code> argument was SQL_ARD_TYPE, and the <code>Col_or_Param_Num</code> descriptor record in the ARD failed the consistency check.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Message</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>22002</td>
<td>Indicator variable required but not supplied</td>
<td>StrLen_or_IndPtr was a null pointer and NULL data was retrieved.</td>
</tr>
<tr>
<td>22003</td>
<td>Numeric value out of range</td>
<td>Returning the numeric value (as numeric or string) for the column would have caused the whole (as opposed to fractional) part of the number to be truncated. For more information, see Appendix D: Data Types.</td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format</td>
<td>The character column in the result set was bound to a C date, time, or timestamp structure, and the value in the column was an invalid date, time, or timestamp, respectively. For more information, see Appendix D: Data Types.</td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
<td>A value from an arithmetic expression that resulted in division by zero was returned.</td>
</tr>
<tr>
<td>22015</td>
<td>Interval field overflow</td>
<td>Assigning from an exact numeric or interval SQL type to an interval C type caused a loss of significant digits in the leading field. When returning data to an interval C type, there was no representation of the value of the SQL type in the interval C type.</td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast specification</td>
<td>A character column in the result set was returned to a character C buffer, and the column contained a character for which there was no representation in the character set of the buffer. The C type was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>(DM) The function was called without first calling SQLFetch or SQLFetchScroll to position the cursor on the row of data required. (DM) The StatementHandle was in an executed state, but no result set was associated with the StatementHandle. A cursor was open on the StatementHandle and SQLFetch or SQLFetchScroll had been called, but the cursor was positioned before the start of the result set or after the end of the result set.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <code>MessageText</code> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY003</td>
<td>Program type out of range</td>
<td>(DM) The argument <code>TargetType</code> was not a valid data type, <code>SQL_C_DEFAULT</code>, <code>SQL_ARD_TYPE</code> (in case of retrieving column data), or <code>SQL_APD_TYPE</code> (in case of retrieving parameter data). (DM) The argument <code>Col_or_Param_Num</code> was 0, and the argument <code>TargetType</code> was not <code>SQL_C_BOOKMARK</code> for a fixed-length bookmark or <code>SQL_C_VARBOOKMARK</code> for a variable-length bookmark.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code>, and then the function was called again on the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code> from a different thread in a multithread application, and then the function was called again on the <code>StatementHandle</code>.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>(DM) The argument <code>TargetValuePtr</code> was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) The specified <code>StatementHandle</code> was not in an executed state. The function was called without first calling <code>SQLExecDirect</code>, <code>SQLExecute</code> or a catalog function. (DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when the <code>SQLGetData</code> function was called. (DM) An asynchronously executing function (not this one) was called for the <code>StatementHandle</code> and was still executing when this function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for the <code>StatementHandle</code> and returned <code>SQL_NEED_DATA</code>. This function was called before data was sent for all data-at-execution parameters or columns. (DM) The <code>StatementHandle</code> was in an executed state, but no result set was associated with the <code>StatementHandle</code>.</td>
</tr>
</tbody>
</table>
A call to `SQLExecute`, `SQLExecDirect`, or `SQLMoreResults` returned `SQL_PARAM_DATAAVAILABLE`, but `SQLGetData` was called, instead of `SQLParamData`.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length (DM) The value specified for argument <code>BufferLength</code> was less than 0.</td>
</tr>
<tr>
<td></td>
<td>The value specified for argument <code>BufferLength</code> was less than 4, the <code>Col_or_Param_Num</code> argument was set to 0, and the driver was an ODBC 2.x driver.</td>
</tr>
<tr>
<td>HY109</td>
<td>Invalid cursor position The cursor was positioned (by <code>SQLSetPos</code>, <code>SQLFetch</code>, <code>SQLFetchScroll</code>, or <code>SQLBulkOperations</code>) on a row that had been deleted or could not be fetched.</td>
</tr>
<tr>
<td></td>
<td>The cursor was a forward-only cursor, and the rowset size was greater than one.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
</tr>
<tr>
<td></td>
<td>(DM) For more information about suspended state, see <code>SQLEndTran Function</code>.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented The driver or data source does not support use of <code>SQLGetData</code> with multiple rows in <code>SQLFetchScroll</code>. This description does not apply to drivers that return the <code>SQL_GD_BLOCK</code> bitmask for the <code>SQL_GETDATA_EXTENSIONS</code> option in <code>SQLGetInfo</code>.</td>
</tr>
<tr>
<td></td>
<td>The driver or data source does not support the conversion specified by the combination of the <code>TargetType</code> argument and the SQL data type of the corresponding column. This error applies only when the SQL data type of the column was mapped to a driver-specific SQL data type.</td>
</tr>
<tr>
<td></td>
<td>The driver supports only ODBC 2.x, and the argument <code>TargetType</code> was one of the following:</td>
</tr>
<tr>
<td></td>
<td><code>SQL_C_NUMERIC SQL_C_SBIGINT SQL_C_UBIGINT</code></td>
</tr>
<tr>
<td></td>
<td>and any of the interval C data types listed in <code>C Data Types</code> in Appendix D: Data Types.</td>
</tr>
<tr>
<td></td>
<td>The driver only supports ODBC versions prior to 3.50, and the argument <code>TargetType</code> was <code>SQL_C_GUID</code>.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
</tr>
<tr>
<td></td>
<td>(DM) The driver corresponding to the <code>StatementHandle</code> does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
</tr>
<tr>
<td></td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><code>SQLCompleteAsync</code> has not been called to complete the previous asynchronous operation on this handle.</td>
</tr>
<tr>
<td></td>
<td>If the previous function call on the handle returns <code>SQL_STILL_EXECUTING</code> and if notification mode is enabled, <code>SQLCompleteAsync</code> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

**Comments**

`SQLGetData` returns the data in a specified column. `SQLGetData` can be called only after one or more rows have been fetched from the result set by `SQLFetch`, `SQLFetchScroll`, or `SQLExtendedFetch`. If variable-length data is too large to be returned in a single call to `SQLGetData` (due to a limitation in the application), `SQLGetData` can retrieve it in parts. It is possible to bind some columns in a row and call `SQLGetData` for others, although this is subject to some restrictions. For more information, see Getting Long Data.

For information about using `SQLGetData` with streamed output parameters, see Retrieving Output Parameters Using SQLGetData.

**Using SQLGetData**

If the driver does not support extensions to `SQLGetData`, the function can return data only for unbound columns with a number greater than that of the last bound column. Furthermore, within a row of data, the value of the `Col_or_Param_Num` argument in each call to `SQLGetData` must be greater than or equal to the value of `Col_or_Param_Num` in the previous call; that is, data must be retrieved in increasing column number order. Finally, if no extensions are supported, `SQLGetData` cannot be called if the rowset size is greater than 1.

Drivers can relax any of these restrictions. To determine what restrictions a driver relaxes, an application calls `SQLGetInfo` with any of the following SQL_GETDATA_EXTENSIONS options:

- `SQL_GD_OUTPUT_PARAMS` = `SQLGetData` can be called to return output parameter values. For more information, see Retrieving Output Parameters Using SQLGetData.

- `SQL_GD_ANY_COLUMN`. If this option is returned, `SQLGetData` can be called for any unbound column, including those before the last bound column.

- `SQL_GD_ANY_ORDER`. If this option is returned, `SQLGetData` can be called for unbound columns in any order.
- SQL_GD_BLOCK. If this option is returned by SQLGetInfo for the SQL_GETDATA_EXTENSIONS InfoType, the driver supports calls to SQLGetData when the rowset size is greater than 1 and the application can call SQLSetPos with the SQL_POSITION option to position the cursor on the correct row before calling SQLGetData.

- SQL_GD_BOUND. If this option is returned, SQLGetData can be called for bound columns as well as unbound columns.

There are two exceptions to these restrictions and a driver’s ability to relax them. First, SQLGetData should never be called for a forward-only cursor when the rowset size is greater than 1. Second, if a driver supports bookmarks, it must always support the ability to call SQLGetData for column 0, even if it does not allow applications to call SQLGetData for other columns before the last bound column. (When an application is working with an ODBC 2.x driver, SQLGetData will successfully return a bookmark when called with Col_or_Param_Num equal to 0 after a call to SQLFetch, because SQLFetch is mapped by the ODBC 3.x Driver Manager to SQLExtendedFetch with a FetchOrientation of SQL_FETCH_NEXT, and SQLGetData with a Col_or_Param_Num of 0 is mapped by the ODBC 3.x Driver Manager to SQLGetStmtOption with an fOption of SQL_GET_BOOKMARK.)

SQLGetData cannot be used to retrieve the bookmark for a row just inserted by calling SQLBulkOperations with the SQL_ADD option, because the cursor is not positioned on the row. An application can retrieve the bookmark for such a row by binding column 0 before calling SQLBulkOperations with SQL_ADD, in which case SQLBulkOperations returns the bookmark in the bound buffer. SQLFetchScroll can then be called with SQL_FETCH_BOOKMARK to reposition the cursor on that row.

If the TargetType argument is an interval data type, the default interval leading precision (2) and the default interval seconds precision (6), as set in the SQL_DESC_DATETIME_INTERVAL_PRECISION and SQL_DESC_PRECISION fields of the ARD, respectively, are used for the data. If the TargetType argument is an SQL_C_NUMERIC data type, the default precision (driver-defined) and default scale (0), as set in the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the ARD, are used for the data. If any default precision or scale is not appropriate, the application should explicitly set the appropriate descriptor field by a call to SQLSetDescField or SQLSetDescRec. It can set the SQL_DESC_CONCISE_TYPE field to SQL_C_NUMERIC and call SQLGetData with a TargetType argument of SQL_ARD_TYPE, which will cause the precision and scale values in the descriptor fields to be used.

**Note**

In ODBC 2.x, applications set TargetType to SQL_C_DATE, SQL_C_TIME, or SQL_C_TIMESTAMP to indicate that *TargetValuePtr is a date, time, or timestamp structure. In ODBC 3.x, applications set TargetType to SQL_C_TYPE_DATE, SQL_C_TYPE_TIME, or SQL_C_TYPE_TIMESTAMP. The Driver Manager makes appropriate mappings if necessary, based on the application and driver version.

**Retrieving Variable-Length Data in Parts**

SQLGetData can be used to retrieve data from a column that contains variable-length data in parts — that is, when the identifier of the SQL data type of the column is SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR, SQL_WCHAR, SQL_WVARCHAR, SQL_WLONGVARCHAR, SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY, or a driver-specific identifier for a variable-length type.

To retrieve data from a column in parts, the application calls SQLGetData multiple times in succession for the same column. On each call, SQLGetData returns the next part of the data. It is up to the application to reassemble the parts, taking care to remove the null-termination character from
intermediate parts of character data. If there is more data to return or not enough buffer was allocated for the terminating character, SQLGetData returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01004 (Data truncated). When it returns the last part of the data, SQLGetData returns SQL_SUCCESS. Neither SQL_NO_TOTAL nor zero can be returned on the last valid call to retrieve data from a column, because the application would then have no way of knowing how much of the data in the application buffer is valid. If SQLGetData is called after this, it returns SQL_NO_DATA. For more information, see the next section, "Retrieving Data with SQLGetData."

Variable-length bookmarks can be returned in parts by SQLGetData. As with other data, a call to SQLGetData to return variable-length bookmarks in parts will return SQLSTATE 01004 (String data, right truncated) and SQL_SUCCESS_WITH_INFO when there is more data to be returned. This is different than the case when a variable-length bookmark is truncated by a call to SQLFetch or SQLFetchScroll, which returns SQL_ERROR and SQLSTATE 22001 (String data, right truncated).

SQLGetData cannot be used to return fixed-length data in parts. If SQLGetData is called more than one time in a row for a column containing fixed-length data, it returns SQL_NO_DATA for all calls after the first.

Retrieving Streamed Output Parameters

If a driver supports streamed output parameters, an application can call SQLGetData with a small buffer many times to retrieve a large parameter value. For more information about streamed output parameter, see Retrieving Output Parameters Using SQLGetData.

Retrieving Data with SQLGetData

To return data for the specified column, SQLGetData performs the following sequence of steps:

1. Returns SQL_NO_DATA if it has already returned all of the data for the column.

2. Sets *StrLen_or_IndPtr to SQL_NULL_DATA if the data is NULL. If the data is NULL and StrLen_or_IndPtr was a null pointer, SQLGetData returns SQLSTATE 22002 (Indicator variable required but not supplied).

   If the data for the column is not NULL, SQLGetData proceeds to step 3.

3. If the SQL_ATTR_MAX_LENGTH statement attribute is set to a nonzero value, if the column contains character or binary data, and if SQLGetData has not previously been called for the column, the data is truncated to SQL_ATTR_MAX_LENGTH bytes.

   Note

   The SQL_ATTR_MAX_LENGTH statement attribute is intended to reduce network traffic. It is generally implemented by the data source, which truncates the data before returning it across the network. Drivers and data sources are not required to support it. Therefore, to guarantee that data is truncated to a particular size, an application should allocate a buffer of that size and specify the size in the BufferLength argument.

4. Converts the data to the type specified in TargetType. The data is given the default precision and scale for that data type. If TargetType is SQL_ARD_TYPE, the data type in the SQL_DESC_CONCISE_TYPE field of the ARD is used. If TargetType is SQL_ARD_TYPE, the data is given the precision and scale in the SQL_DESC_DATETIME_INTERVAL_PRECISION,
SQL_DESC_PRECISION, and SQL_DESC_SCALE fields of the ARD, depending on the data type in the SQL_DESC_CONCISE_TYPE field. If any default precision or scale is not appropriate, the application should explicitly set the appropriate descriptor field by a call to SQLSetDescField or SQLSetDescRec.

5. If the data was converted to a variable-length data type, such as character or binary, SQLGetData checks whether the length of the data exceeds BufferLength. If the length of character data (including the null-termination character) exceeds BufferLength, SQLGetData truncates the data to BufferLength less the length of a null-termination character. It then null-terminates the data. If the length of binary data exceeds the length of the data buffer, SQLGetData truncates it to BufferLength bytes.

If the data buffer supplied is too small to hold the null-termination character, SQLGetData returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01004.

SQLGetData never truncates data converted to fixed-length data types; it always assumes that the length of *TargetValuePtr is the size of the data type.

6. Places the converted (and possibly truncated) data in *TargetValuePtr. Note that SQLGetData cannot return data out of line.

7. Places the length of the data in *StrLen_or_IndPtr. If StrLen_or_IndPtr was a null pointer, SQLGetData does not return the length.

- For character or binary data, this is the length of the data after conversion and before truncation due to BufferLength. If the driver cannot determine the length of the data after conversion, as is sometimes the case with long data, it returns SQL_SUCCESS_WITH_INFO and sets the length to SQL_NO_TOTAL. (The last call to SQLGetData must always return the length of the data, not zero or SQL_NO_TOTAL.) If data was truncated due to the SQL_ATTR_MAX_LENGTH statement attribute, the value of this attribute — as opposed to the actual length — is placed in *StrLen_or_IndPtr. This is because this attribute is designed to truncate data on the server before conversion, so the driver has no way of figuring out what the actual length is. When SQLGetData is called multiple times in succession for the same column, this is the length of the data available at the start of the current call; that is, the length decreases with each subsequent call.

- For all other data types, this is the length of the data after conversion; that is, it is the size of the type to which the data was converted.

8. If the data is truncated without loss of significance during conversion (for example, the real number 1.234 is truncated when converted to the integer 1) or because BufferLength is too small (for example, the string "abcdef" is placed in a 4-byte buffer), SQLGetData returns SQLSTATE 01004 (Data truncated) and SQL_SUCCESS_WITH_INFO. If data is truncated without loss of significance due to the SQL_ATTR_MAX_LENGTH statement attribute, SQLGetData returns SQL_SUCCESS and does not return SQLSTATE 01004 (Data truncated).

The contents of the bound data buffer (if SQLGetData is called on a bound column) and the length/indicator buffer are undefined if SQLGetData does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

Successive calls to SQLGetData will retrieve data from the last column requested; prior offsets become invalid. For example, when the following sequence is performed:

```
SQLGetData(icol=n), SQLGetData(icol=m), SQLGetData(icol=n)
```
the second call to SQLGetData(icol=n) retrieves data from the start of the n column. Any offset in the data due to earlier calls to SQLGetData for the column is no longer valid.

Descriptors and SQLGetData

SQLGetData does not interact directly with any descriptor fields.

If TargetType is SQL_ARD_TYPE, the data type in the SQL_DESC_CONCISE_TYPE field of the ARD is used. If TargetType is either SQL_ARD_TYPE or SQL_C_DEFAULT, the data is given the precision and scale in the SQL_DESC_DATETIME_INTERVAL_PRECISION, SQL_DESC_PRECISION, and SQL_DESC_SCALE fields of the ARD, depending on the data type in the SQL_DESC_CONCISE_TYPE field.

Code Example

In the following example, an application executes a SELECT statement to return a result set of the customer IDs, names, and phone numbers sorted by name, ID, and phone number. For each row of data, it calls SQLFetch to position the cursor to the next row. It calls SQLGetData to retrieve the fetched data; the buffers for the data and the returned number of bytes are specified in the call to SQLGetData. Finally, it prints each employee's name, ID, and phone number.

```c
#define NAME_LEN 50
#define PHONE_LEN 50

SQLCHAR szName[NAME_LEN], szPhone[PHONE_LEN];
SQLINTEGER sCustID, cbName, cbAge, cbBirthday;
SQLRETURN retcode;
SQLHSTMT hstmt;

retcode = SQLExecDirect(hstmt,
    "SELECT CUSTID, NAME, PHONE FROM CUSTOMERS ORDER BY 2, 1, 3",
    SQL_NTS);

if (retcode == SQL_SUCCESS) {
    while (TRUE) {
        retcode = SQLFetch(hstmt);
        if (retcode == SQL_ERROR || retcode == SQL_SUCCESS_WITH_INFO) {
            show_error();
        } else if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){
            /* Get data for columns 1, 2, and 3 */
            SQLGetData(hstmt, 1, SQL_C_ULONG, &sCustID, 0, &cbCustID);
            SQLGetData(hstmt, 2, SQL_C_CHAR, szName, NAME_LEN, &cbName);
            SQLGetData(hstmt, 3, SQL_C_CHAR, szPhone, PHONE_LEN, &cbPhone);
            /* Print the row of data */
            fprintf(out, "%-5d %-*s %*s", sCustID, NAME_LEN-1, szName,
```
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assigning storage for a column in a result set</td>
<td>SQLBindCol</td>
</tr>
<tr>
<td>Performing bulk operations that do not relate to the block cursor position</td>
<td>SQLBulkOperations</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td>Fetching a single row of data or a block of data in a forward-only direction</td>
<td>SQLFetch</td>
</tr>
<tr>
<td>Sending parameter data at execution time</td>
<td>SQLPutData</td>
</tr>
<tr>
<td>Positioning the cursor, refreshing data in the rowset, or updating or deleting data in the rowset</td>
<td>SQLSetPos</td>
</tr>
</tbody>
</table>

See Also

- ODBC API Reference
- ODBC Header Files
- Retrieving Output Parameters Using SQLGetData

© 2016 Microsoft
Summary

**SQLGetDescField** returns the current setting or value of a single field of a descriptor record.

Syntax

```c
SQLRETURN SQLGetDescField(
    SQLHDESC    DescriptorHandle,
    SQLSMALLINT RecNumber,
    SQLSMALLINT FieldIdentifier,
    SQLPOINTER  ValuePtr,
    SQLINTEGER  BufferLength,
    SQLINTEGER * StringLengthPtr);
```

Arguments

*DescriptorHandle*

[Input] Descriptor handle.

*RecNumber*

[Input] Indicates the descriptor record from which the application seeks information. Descriptor records are numbered from 0, with record number 0 being the bookmark record. If the *FieldIdentifier* argument indicates a header field, *RecNumber* is ignored. If *RecNumber* is less than or equal to SQL_DESC_COUNT but the row does not contain data for a column or parameter, a call to **SQLGetDescField** will return the default values of the fields. (For more information, see "Initialization of Descriptor Fields" in **SQLSetDescField**.)

*FieldIdentifier*

[Input] Indicates the field of the descriptor whose value is to be returned. For more information, see the "*FieldIdentifier* Argument" section in **SQLSetDescField**.

*ValuePtr*

[Output] Pointer to a buffer in which to return the descriptor information. The data type depends on the value of *FieldIdentifier*.

If *ValuePtr* is integer type, applications should use a buffer of SQLULEN and initialize the value to 0 before calling this function as some drivers may only write the lower 32-bit or 16-bit of a buffer and leave the higher-order bit unchanged.

If *ValuePtr* is NULL, *StringLengthPtr* will still return the total number of bytes (excluding the null-termination character for character data) available to return in the buffer pointed to by *ValuePtr*.

*BufferLength*

[Input] If *FieldIdentifier* is an ODBC-defined field and *ValuePtr* points to a character string or a binary buffer, this argument should be the length of *ValuePtr*. If *FieldIdentifier* is an ODBC-defined field and *ValuePtr* is an integer, *BufferLength* is ignored. If the value in *ValuePtr* is of a Unicode data type (when calling **SQLGetDescFieldW**), the *BufferLength* argument must be an even number.

If *FieldIdentifier* is a driver-defined field, the application indicates the nature of the field to the Driver Manager by setting the *BufferLength* argument. *BufferLength* can have the following
If *ValuePtr is a pointer to a character string, then BufferLength is the length of the string or SQL_NTS.

If *ValuePtr is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in BufferLength. This places a negative value in BufferLength.

If *ValuePtr is a pointer to a value other than a character string or binary string, then BufferLength should have the value SQL_IS_POINTER.

If *ValuePtr contains a fixed-length data type, then BufferLength is either SQL_IS_INTEGER, SQL_IS_UINTEGER, SQL_IS_SMALLINT, or SQL_IS_USMALLINT, as appropriate.

**StringLengthPtr**

[Output] Pointer to the buffer in which to return the total number of bytes (excluding the number of bytes required for the null-termination character) available to return in *ValuePtr.

### Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_NO_DATA, or SQL_INVALID_HANDLE.

SQL_NO_DATA is returned if RecNumber is greater than the current number of descriptor records.

SQL_NO_DATA is returned if DescriptorHandle is an IRD handle and the statement is in the prepared or executed state but there was no open cursor associated with it.

### Diagnostics

When SQLGetDescField returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLGetDescField and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The buffer *ValuePtr was not large enough to return the entire descriptor field, so the field was truncated. The length of the untruncated descriptor field is returned in *StringLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>(DM) The RecNumber argument was equal to 0, the SQL_ATTR_USE_BOOKMARK statement attribute was SQL_UB_OFF, and the DescriptorHandle argument was an IRD handle. (This error can be returned for an explicitly allocated descriptor only if the descriptor is associated with a statement handle.) The FieldIdentifier argument was a record field, the RecNumber argument was 0, and the DescriptorHandle argument was an IPD handle. The RecNumber argument was less than 0.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate the memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY007</td>
<td>Associated statement is not prepared</td>
<td>DescriptorHandle was associated with a StatementHandle as an IRD, and the associated statement handle had not been prepared or executed.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) DescriptorHandle was associated with a StatementHandle for which an asynchronously executing function (not this one) was called and was still executing when this function was called. (DM) DescriptorHandle was associated with a StatementHandle for which SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) An asynchronously executing function was called for the connection handle that is associated with the DescriptorHandle. This asynchronous function was still executing when the SQLGetDescField function was called.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY021</td>
<td>Inconsistent descriptor information</td>
<td>The SQL_DESC_TYPE and SQL_DESC_DATETIME_INTERVAL_CODE fields do not form a valid ODBC SQL type, a valid driver-specific SQL type (for IPDs), or a valid ODBC C type (for APDs or ARDs).</td>
</tr>
</tbody>
</table>
**HY090** Invalid string or buffer length

(DM) *ValuePtr was a character string, and BufferLength was less than zero.

**HY091** Invalid descriptor field identifier

*FieldIdentifier was not an ODBC-defined field and was not an implementation-defined value.*

*FieldIdentifier was undefined for the DescriptorHandle.*

**HY117** Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.

(DM) For more information about the suspended state, see SQLEndTran Function.

**HYT01** Connection timeout expired

The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.

**IM001** Driver does not support this function

(DM) The driver associated with the DescriptorHandle does not support the function.

---

**Comments**

An application can call SQLGetDescField to return the value of a single field of a descriptor record. A call to SQLGetDescField can return the setting of any field in any descriptor type, including header fields, record fields, and bookmark fields. An application can obtain the settings of multiple fields in the same or different descriptors, in arbitrary order, by making repeated calls to SQLGetDescField. SQLGetDescField can also be called to return driver-defined descriptor fields.

For performance reasons, an application should not call SQLGetDescField for an IRD before executing a statement.

The settings of multiple fields that describe the name, data type, and storage of column or parameter data can also be retrieved in a single call to SQLGetDescRec. SQLGetStmtAttr can be called to return the setting of a single field in the descriptor header that is also a statement attribute. SQLColAttribute, SQLDescribeCol, and SQLDescribeParam return record or bookmark fields.

When an application calls SQLGetDescField to retrieve the value of a field that is undefined for a particular descriptor type, the function returns SQL_SUCCESS but the value returned for the field is undefined. For example, calling SQLGetDescField for the SQL_DESC_NAME or SQL_DESC_NULLABLE field of an APD or ARD will return SQL_SUCCESS but an undefined value for the field.

When an application calls SQLGetDescField to retrieve the value of a field that is defined for a particular descriptor type but that has no default value and has not been set yet, the function returns SQL_SUCCESS but the value returned for the field is undefined. For more information on the initialization of descriptor fields and descriptions of the fields, see "Initialization of Descriptor Fields" in SQLSetDescField. For more information on descriptors, see Descriptors.
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting multiple descriptor fields</td>
<td>SQLGetDescRec Function</td>
</tr>
<tr>
<td>Setting a single descriptor field</td>
<td>SQLSetDescField Function</td>
</tr>
<tr>
<td>Setting multiple descriptor fields</td>
<td>SQLSetDescRec Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLGetDescRec Function

Conformance
Version Introduced: ODBC 3.0
Standards Compliance: ISO 92

Summary
SQLGetDescRec returns the current settings or values of multiple fields of a descriptor record. The fields returned describe the name, data type, and storage of column or parameter data.

Syntax

```c
SQLRETURN SQLGetDescRec(
    SQLHDESC  DescriptorHandle,
    SQLSMALLINT  RecNumber,
    SQLCHAR *  Name,
    SQLSMALLINT  BufferLength,
    SQLSMALLINT *  StringLengthPtr,
    SQLSMALLINT *  TypePtr,
    SQLSMALLINT *  SubTypePtr,
    SQLLEN *  LengthPtr,
    SQLSMALLINT *  PrecisionPtr,
    SQLSMALLINT *  ScalePtr,
    SQLSMALLINT *  NullablePtr);
```
Arguments

**DescriptorHandle**

[Input] Descriptor handle.

**RecNumber**

[Input] Indicates the descriptor record from which the application seeks information. Descriptor records are numbered from 1, with record number 0 being the bookmark record. The **RecNumber** argument must be less than or equal to the value of SQL_DESC_COUNT. If **RecNumber** is less than or equal to SQL_DESC_COUNT but the row does not contain data for a column or parameter, a call to SQLGetDescRec will return the default values of the fields. (For more information, see "Initialization of Descriptor Fields" in SQLSetDescField.)

**Name**

[Output] A pointer to a buffer in which to return the SQL_DESC_NAME field for the descriptor record.

If **Name** is NULL, **StringLengthPtr** will still return the total number of characters (excluding the null-termination character for character data) available to return in the buffer pointed to by **Name**.

**BufferLength**

[Input] Length of the *Name buffer, in characters.

**StringLengthPtr**

[Output] A pointer to a buffer in which to return the number of characters of data available to return in the *Name buffer, excluding the null-termination character. If the number of characters was greater than or equal to BufferLength, the data in *Name is truncated to BufferLength minus the length of a null-termination character, and is null-terminated by the driver.

**TypePtr**

[Output] A pointer to a buffer in which to return the value of the SQL_DESC_TYPE field for the descriptor record.

**SubTypePtr**

[Output] For records whose type is SQL_DATETIME or SQL_INTERVAL, this is a pointer to a buffer in which to return the value of the SQL_DESC_DATETIME_INTERVAL_CODE field.

**LengthPtr**

[Output] A pointer to a buffer in which to return the value of the SQL_DESC_OCTET_LENGTH field for the descriptor record.

**PrecisionPtr**

[Output] A pointer to a buffer in which to return the value of the SQL_DESC_PRECISION field for the descriptor record.

**ScalePtr**

[Output] A pointer to a buffer in which to return the value of the SQL_DESC_SCALE field for the descriptor record.

**NullablePtr**

[Output] A pointer to a buffer in which to return the value of the SQL_DESC_NULLABLE field for the descriptor record.

Returns
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_NO_DATA, or SQL_INVALID_HANDLE.

SQL_NO_DATA is returned if RecNumber is greater than the current number of descriptor records.

SQL_NO_DATA is returned if DescriptorHandle is an IRD handle and the statement is in the prepared or executed state but there was no open cursor associated with it.

**Diagnostics**

When SQLGetDescRec returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_DESC and a Handle of DescriptorHandle. The following table lists the SQLSTATE values typically returned by SQLGetDescRec and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The buffer *Name was not large enough to return the entire descriptor field. Therefore, the field was truncated. The length of the untruncated descriptor field is returned in *StringLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
</tbody>
</table>
| 07009    | Invalid descriptor index | The FieldIdentifier argument was a record field, the RecNumber argument was set to 0, and the DescriptorHandle argument was an IPD handle.

(DM) The RecNumber argument was set to 0, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF, and the DescriptorHandle argument was an IRD handle.

The RecNumber argument was less than 0. |
| 08S01    | Communication link failure | The communication link between the driver and the data source to which the driver was connected failed before the function completed processing. |
| HY000    | General error | An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause. |
| HY001    | Memory allocation error | The driver was unable to allocate the memory that is required to support execution or completion of the function. |
| HY007    | Associated statement is not prepared | DescriptorHandle was associated with an IRD, and the associated statement handle was not in the prepared or
executed state.

| HY010 | Function sequence error | (DM) DescriptorHandle was associated with a StatementHandle for which an asynchronously executing function (not this one) was called and was still executing when this function was called.

(DM) DescriptorHandle was associated with a StatementHandle for which SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

(DM) An asynchronously executing function was called for the connection handle that is associated with the DescriptorHandle. This asynchronous function was still executing when SQLGetDescRec was called.

| HY013 | Memory management error | The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.

| HY117 | Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed. | (DM) For more information about suspended state, see SQLEndTran Function.

| HYT01 | Connection timeout expired | The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.

| IM001 | Driver does not support this function | (DM) The driver associated with DescriptorHandle does not support the function.

**Comments**

An application can call SQLGetDescRec to retrieve the values of the following descriptor fields for a single column or parameter:

- SQL_DESC_NAME
- SQL_DESC_TYPE
- SQL_DESC_DATETIME_INTERVAL_CODE (for records whose type is SQL_DATETIME or SQL_INTERVAL)
- SQL_DESC_OCTET_LENGTH
SQLGetDescRec does not retrieve the values for header fields.

An application can prevent the return of a field’s setting by setting the argument that corresponds to the field to a null pointer.

When an application calls SQLGetDescRec to retrieve the value of a field that is undefined for a particular descriptor type, the function returns SQL_SUCCESS but the value returned for the field is undefined. For example, calling SQLGetDescRec for the SQL_DESC_NAME or SQL_DESC_NULLABLE field of an APD or ARD will return SQL_SUCCESS but an undefined value for the field.

When an application calls SQLGetDescRec to retrieve the value of a field that is defined for a particular descriptor type but that has no default value and has not been set yet, the function returns SQL_SUCCESS but the value returned for the field is undefined. For more information, see "Initialization of Descriptor Fields" in SQLSetDescField.

The values of fields can also be retrieved individually by a call to SQLGetDescField. For a description of the fields in a descriptor header or record, see SQLSetDescField. For more information about descriptors, see Descriptors.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a column</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Binding a parameter</td>
<td>SQLBindParameter Function</td>
</tr>
<tr>
<td>Getting a descriptor field</td>
<td>SQLGetDescField Function</td>
</tr>
<tr>
<td>Setting multiple descriptor fields</td>
<td>SQLSetDescRec Function</td>
</tr>
</tbody>
</table>

**See Also**

- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft
SQLGetDiagField

**Conformance**
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

**Summary**
SQLGetDiagField returns the current value of a field of a record of the diagnostic data structure (associated with a specified handle) that contains error, warning, and status information.

**Syntax**

```c
SQLRETURN SQLGetDiagField(
    SQLSMALLINT HandleType,
    SQLHANDLE Handle,
    SQLSMALLINT RecNumber,
    SQLSMALLINT DiagIdentifier,
    SQLPOINTER DiagInfoPtr,
    SQLSMALLINT BufferLength,
    SQLSMALLINT *StringLengthPtr);
```

**Arguments**

*HandleType*
[Input] A handle type identifier that describes the type of handle for which diagnostics are required. Must be one of the following:

- SQL_HANDLE_DBC
- SQL_HANDLE_DBC_INFO_TOKEN
- SQL_HANDLE_DESC
- SQL_HANDLE_ENV
- SQL_HANDLE_STMT

If HandleType is SQL_HANDLE_DBC_INFO_TOKEN, handle is used only by the Driver Manager and driver. Applications should not use this handle type. For more information about SQL_HANDLE_DBC_INFO_TOKEN, see Developing Connection-Pool Awareness in an ODBC Driver.

*Handle*
[Input] A handle for the diagnostic data structure, of the type indicated by HandleType. If HandleType is SQL_HANDLE_ENV, Handle can be either a shared or an unshared environment handle.

*RecNumber*
[Input] Indicates the status record from which the application seeks information. Status records are numbered from 1. If the DiagIdentifier argument indicates any field of the diagnostics header, RecNumber is ignored. If not, it should be more than 0.

*DiagIdentifier*

[Input] Indicates the field of the diagnostic whose value is to be returned. For more information, see the "DiagIdentifier Argument" section in "Comments."

**DiagInfoPtr**

[Output] Pointer to a buffer in which to return the diagnostic information. The data type depends on the value of DiagIdentifier. If DiagInfoPtr is an integer type, applications should use a buffer of SQLULEN and initialize the value to 0 before calling this function, as some drivers may only write the lower 32-bit or 16-bit of a buffer and leave the higher-order bit unchanged.

If DiagInfoPtr is NULL, StringLengthPtr will still return the total number of bytes (excluding the null-termination character for character data) available to return in the buffer pointed to by DiagInfoPtr.

**BufferLength**

[Input] If DiagIdentifier is an ODBC-defined diagnostic and DiagInfoPtr points to a character string or a binary buffer, this argument should be the length of *DiagInfoPtr. If DiagIdentifier is an ODBC-defined field and *DiagInfoPtr is an integer, BufferLength is ignored. If the value in *DiagInfoPtr is a Unicode string (when calling SQLGetDiagFieldW), the BufferLength argument must be an even number.

If DiagIdentifier is a driver-defined field, the application indicates the nature of the field to the Driver Manager by setting the BufferLength argument. BufferLength can have the following values:

- If DiagInfoPtr is a pointer to a character string, BufferLength is the length of the string or SQL_NTS.
- If DiagInfoPtr is a pointer to a binary buffer, the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in BufferLength. This places a negative value in BufferLength.
- If DiagInfoPtr is a pointer to a value other than a character string or binary string, BufferLength should have the value SQL_IS_POINTER.
- If *DiagInfoPtr contains a fixed-length data type, BufferLength is SQL_IS_INTEGER, SQL_IS_UINT, SQL_IS_SMALLINT, or SQL_IS_USMALLINT, as appropriate.

**StringLengthPtr**

[Output] Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes required for the null-termination character) available to return in *DiagInfoPtr, for character data. If the number of bytes available to return is greater than or equal to BufferLength, the text in *DiagInfoPtr is truncated to BufferLength minus the length of a null-termination character.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_NO_DATA.

**Diagnostics**

SQLGetDiagField does not post diagnostic records for itself. It uses the following return values to
report the outcome of its own execution:

- **SQL_SUCCESS**: The function successfully returned diagnostic information.
- **SQL_SUCCESS_WITH_INFO**: \*DiagInfoPtr was too small to hold the requested diagnostic field. Therefore, the data in the diagnostic field was truncated. To determine that a truncation occurred, the application must compare BufferLength to the actual number of bytes available, which is written to \*StringLengthPtr.
- **SQL_INVALID_HANDLE**: The handle indicated by HandleType and Handle was not a valid handle.
- **SQL_ERROR**: One of the following occurred:
  - The DiagIdentifier argument was not one of the valid values.
  - The DiagIdentifier argument was SQL_DIAG_CURSOR_ROW_COUNT, SQL_DIAG_DYNAMIC_FUNCTION, SQL_DIAG_DYNAMIC_FUNCTION_CODE, or SQL_DIAG_ROW_COUNT, but Handle was not a statement handle. (The Driver Manager returns this diagnostic.)
  - The RecNumber argument was negative or 0 when DiagIdentifier indicated a field from a diagnostic record. RecNumber is ignored for header fields.
  - The value requested was a character string and BufferLength was less than zero.
  - When using asynchronous notification, the asynchronous operation on the handle was not complete.
- **SQL_NO_DATA**: RecNumber was greater than the number of diagnostic records that existed for the handle specified in Handle. The function also returns SQL_NO_DATA for any positive RecNumber if there are no diagnostic records for Handle.

Comments

An application typically calls SQLGetDiagField to accomplish one of three goals:

1. To obtain specific error or warning information when a function call has returned SQL_ERROR or SQL_SUCCESS_WITH_INFO (or SQL_NEED_DATA for the SQLBrowseConnect function).
2. To determine the number of rows in the data source that were affected when insert, delete, or update operations were performed with a call to SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos (from the SQL_DIAG_ROW_COUNT header field), or to determine the number of rows that exist in the current open cursor, if the driver can provide this information (from the SQL_DIAG_CURSOR_ROW_COUNT header field).
3. To determine which function was executed by a call to SQLExecDirect or SQLExecute (from the SQL_DIAG_DYNAMIC_FUNCTION and SQL_DIAG_DYNAMIC_FUNCTION_CODE header fields).

Any ODBC function can post zero or more diagnostic records every time that it is called, so an application can call SQLGetDiagField after any ODBC function call. There is no limit to the number of diagnostic records that can be stored at any one time. SQLGetDiagField retrieves only the diagnostic information most recently associated with the diagnostic data structure specified in the Handle.
argument. If the application calls an ODBC function other than SQLGetDiagField or SQLGetDiagRec, any diagnostic information from a previous call with the same handle is lost.

An application can scan all diagnostic records by incrementing RecNumber, as long as SQLGetDiagField returns SQL_SUCCESS. The number of status records is indicated in the SQL_DIAG_NUMBER header field. Calls to SQLGetDiagField are nondestructive to the header and record fields. The application can call SQLGetDiagField again later to retrieve a field from a record, as long as a function other than the diagnostic functions has not been called in the interim, which would post records on the same handle.

An application can call SQLGetDiagField to return any diagnostic field at any time, except for SQL_DIAG_CURSOR_ROW_COUNT or SQL_DIAG_ROW_COUNT, which will return SQL_ERROR if Handle is not a statement handle. If any other diagnostic field is undefined, the call to SQLGetDiagField will return SQL_SUCCESS (provided no other diagnostic is encountered) and an undefined value is returned for the field.

For more information, see Using SQLGetDiagRec and SQLGetDiagField and Implementing SQLGetDiagRec and SQLGetDiagField.

Calling an API other than the one that’s being executed asynchronously will generate HY010 "Function sequence error". However, the error record cannot be retrieved before the asynchronous operation completes.

**HandleType Argument**

Each handle type can have diagnostic information associated with it. The HandleType argument indicates the handle type of Handle.

Some header and record fields cannot be returned for environment, connection, statement, and descriptor handles. Those handles for which a field is not applicable are indicated in the "Header Fields" and "Record Fields" sections following.

If HandleType is SQL_HANDLE_ENV, Handle can be either a shared or unshared environment handle.

No driver-specific header diagnostic fields should be associated with an environment handle.

The only diagnostic header fields that are defined for a descriptor handle are SQL_DIAG_NUMBER and SQL_DIAG_RETURNCODE.

**DiagIdentifier Argument**

This argument indicates the identifier of the field required from the diagnostic data structure. If RecNumber is greater than or equal to 1, the data in the field describes the diagnostic information returned by a function. If RecNumber is 0, the field is in the header of the diagnostic data structure and therefore contains data pertaining to the function call that returned the diagnostic information, not to the specific information.

Drivers can define driver-specific header and record fields in the diagnostic data structure.

An ODBC 3.x application working with an ODBC 2.x driver will be able to call SQLGetDiagField only with a DiagIdentifier argument of SQL_DIAG_CLASS_ORIGIN, SQL_DIAG_CLASS_SUBCLASS_ORIGIN, SQL_DIAG_CONNECTION_NAME, SQL_DIAG_MESSAGE_TEXT, SQL_DIAG_NATIVE, SQL_DIAG_NUMBER, SQL_DIAG_RETURNCODE, SQL_DIAG_SERVER_NAME, or SQL_DIAG_SQLSTATE.
All other diagnostic fields will return SQL_ERROR.

**Header Fields**

The header fields listed in the following table can be included in the `DiagIdentifier` argument.

<table>
<thead>
<tr>
<th><code>DiagIdentifier</code></th>
<th>Return type</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DIAG_CURSOR_ROW_COUNT</td>
<td>SQLLEN</td>
<td>This field contains the count of rows in the cursor. Its semantics depend on the <code>SQLGetInfo</code> information types <code>SQL_DYNAMIC_CURSOR_ATTRIBUTES2</code>, <code>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2</code>, <code>SQL_KEYSET_CURSOR_ATTRIBUTES2</code>, and <code>SQL_STATIC_CURSOR_ATTRIBUTES2</code>, which indicate which row counts are available for each cursor type (in the <code>SQL_CA2_CRC_EXACT</code> and <code>SQL_CA2_CRC_APPROXIMATE</code> bits). The contents of this field are defined only for statement handles and only after <code>SQLExecute</code>, <code>SQLExceptionDirect</code>, or <code>SQLMoreResults</code> has been called. Calling <code>SQLGetDiagField</code> with a <code>DiagIdentifier</code> of <code>SQL_DIAG_CURSOR_ROW_COUNT</code> on other than a statement handle will return SQL_ERROR.</td>
</tr>
<tr>
<td>SQL_DIAG_DYNAMIC_FUNCTION</td>
<td>SQLCHAR *</td>
<td>This is a string that describes the SQL statement that the underlying function executed. (See &quot;Values of the Dynamic Function fields,&quot; later in this section, for specific values.) The contents of this field are defined only for statement handles and only after a call to <code>SQLExecute</code>, <code>SQLExceptionDirect</code>, or <code>SQLMoreResults</code>. Calling <code>SQLGetDiagField</code> with a <code>DiagIdentifier</code> of <code>SQL_DIAG_DYNAMIC_FUNCTION</code> on other than a statement handle will return SQL_ERROR. The value of this field is undefined before a call to <code>SQLExecute</code> or <code>SQLExceptionDirect</code>.</td>
</tr>
<tr>
<td>SQL_DIAG_DYNAMIC_FUNCTION_CODE</td>
<td>SQLINTEGER</td>
<td>This is a numeric code that describes the SQL statement that was executed by the underlying function. (See &quot;Values of the Dynamic Function Fields,&quot; later in this section, for specific value.) The contents of this field are defined only for statement handles and only after a call to <code>SQLExecute</code>, <code>SQLExceptionDirect</code>, or <code>SQLMoreResults</code>. Calling <code>SQLGetDiagField</code> with a <code>DiagIdentifier</code> of <code>SQL_DIAG_DYNAMIC_FUNCTION_CODE</code> on other than a statement handle will return SQL_ERROR.</td>
</tr>
</tbody>
</table>

The value of this field is undefined before a call to `SQLExecute` or `SQLExceptionDirect`. |
SQL_Diag_DYNAMIC_FUNCTION_CODE on other than a statement handle will return SQL_ERROR. The value of this field is undefined before a call to SQLExecute or SQLExecDirect.

<table>
<thead>
<tr>
<th>DiagIdentifier</th>
<th>Return type</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DIAG_NUMBER</td>
<td>SQLINTEGER</td>
<td>The number of status records that are available for the specified handle.</td>
</tr>
<tr>
<td>SQL_DIAG_RETURNCODE</td>
<td>SQLRETURN</td>
<td>Return code returned by the function. For a list of return codes, see Return Codes. The driver does not have to implement SQL_DIAG_RETURNCODE; it is always implemented by the Driver Manager. If no function has yet been called on the Handle, SQL_SUCCESS will be returned for SQL_DIAG.ReturnCODE.</td>
</tr>
<tr>
<td>SQL_DIAG_ROW_COUNT</td>
<td>SQLLEN</td>
<td>The number of rows affected by an insert, delete, or update performed by SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos. It is driver-defined after a cursor specification has been executed. The contents of this field are defined only for statement handles. Calling SQLGetDiagField with a DiagIdentifier of SQL_DIAG_ROW_COUNT on other than a statement handle will return SQL_ERROR. The data in this field is also returned in the RowCountPtr argument of SQLRowCount. The data in this field is reset after every nondiagnostic function call, whereas the row count returned by SQLRowCount remains the same until the statement is set back to the prepared or allocated state.</td>
</tr>
</tbody>
</table>

Record Fields

The record fields listed in the following table can be included in the DiagIdentifier argument.

<table>
<thead>
<tr>
<th>DiagIdentifier</th>
<th>Return type</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DIAG_CLASS_ORIGIN</td>
<td>SQLCHAR *</td>
<td>A string that indicates the document that defines the class portion of the SQLSTATE value in this record. Its value is &quot;ISO 9075&quot; for all SQLSTATEs defined by Open Group and ISO call-level interface. For ODBC-specific SQLSTATEs (all those whose SQLSTATE class is &quot;IM&quot;), its value is &quot;ODBC 3.0&quot;.</td>
</tr>
<tr>
<td>Field Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SQL_DIAG_COLUMN_NUMBER</td>
<td>SQLINTEGER</td>
<td>If the SQL_DIAG_ROW_NUMBER field is a valid row number in a rowset or a set of parameters, this field contains the value that represents the column number in the result set or the parameter number in the set of parameters. Result set column numbers always start at 1; if this status record pertains to a bookmark column, the field can be zero. Parameter numbers start at 1. It has the value SQL_NO_COLUMN_NUMBER if the status record is not associated with a column number or parameter number. If the driver cannot determine the column number or parameter number that this record is associated with, this field has the value SQL_COLUMN_NUMBER_UNKNOWN. The contents of this field are defined only for statement handles.</td>
</tr>
<tr>
<td>SQL_DIAG_CONNECTION_NAME</td>
<td>SQLCHAR *</td>
<td>A string that indicates the name of the connection that the diagnostic record relates to. This field is driver-defined. For diagnostic data structures associated with the environment handle and for diagnostics that do not relate to any connection, this field is a zero-length string.</td>
</tr>
<tr>
<td>SQL_DIAG_MESSAGE_TEXT</td>
<td>SQLCHAR *</td>
<td>An informational message on the error or warning. This field is formatted as described in Diagnostic Messages. There is no maximum length to the diagnostic message text.</td>
</tr>
<tr>
<td>SQL_DIAG_NATIVE</td>
<td>SQLINTEGER</td>
<td>A driver/data source–specific native error code. If there is no native error code, the driver returns 0.</td>
</tr>
<tr>
<td>SQL_DIAG_ROW_NUMBER</td>
<td>SQLLEN</td>
<td>This field contains the row number in the rowset, or the parameter number in the set of parameters, with which the status record is associated. Row numbers and parameter numbers start with 1. This field has the value SQL_NO_ROW_NUMBER if this status record is not associated with a row number or parameter number. If the driver cannot determine the row number or parameter number that this record is associated with, this field has the value SQL_ROW_NUMBER_UNKNOWN. The contents of this field are defined only for statement handles.</td>
</tr>
<tr>
<td>SQL_DIAG_SERVER_NAME</td>
<td>SQLCHAR *</td>
<td>A string that indicates the server name that the diagnostic record relates to. It is the same as the value returned for a call to SQLGetInfo with the</td>
</tr>
</tbody>
</table>
SQL_DATA_SOURCE_NAME option. For diagnostic data structures associated with the environment handle and for diagnostics that do not relate to any server, this field is a zero-length string.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DIAG_SQLSTATE</td>
<td>SQLCHAR *</td>
<td>A five-character SQLSTATE diagnostic code. For more information, see SQLSTATEs.</td>
</tr>
<tr>
<td>SQL_DIAG_SUBCLASS_ORIGIN</td>
<td>SQLCHAR *</td>
<td>A string with the same format and valid values as SQL_DIAG_CLASS_ORIGIN, that identifies the defining portion of the subclass portion of the SQLSTATE code. The ODBC-specific SQLSTATEs for which &quot;ODBC 3.0&quot; is returned include the following: 01S00, 01S01, 01S02, 01S06, 01S07, 07S01, 08S01, 21S01, 21S02, 25S01, 25S02, 25S03, 42S01, 42S02, 42S11, 42S12, 42S21, 42S22, HY095, HY097, HY098, HY099, HY100, HY101, HY105, HY107, HY109, HY110, HY111, HYT00, HYT01, IM001, IM002, IM003, IM004, IM005, IM006, IM007, IM008, IM010, IM011, IM012.</td>
</tr>
</tbody>
</table>

Values of the Dynamic Function Fields

The following table describes the values of SQL_DIAG_DYNAMIC_FUNCTION and SQL_DIAG_DYNAMIC_FUNCTION_CODE that apply to each type of SQL statement executed by a call to SQLExecute or SQLExecDirect. The driver can add driver-defined values to those listed.

<table>
<thead>
<tr>
<th>SQL statement executed</th>
<th>Value of SQL_DIAG_DYNAMIC_FUNCTION</th>
<th>Value of SQL_DIAG_DYNAMIC_FUNCTION_CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>alter-domain-statement</td>
<td>&quot;ALTER DOMAIN&quot;</td>
<td>SQL_DIAG_ALTER_DOMAIN</td>
</tr>
<tr>
<td>alter-table-statement</td>
<td>&quot;ALTER TABLE&quot;</td>
<td>SQL_DIAG_ALTER_TABLE</td>
</tr>
<tr>
<td>assertion-definition</td>
<td>&quot;CREATE ASSERTION&quot;</td>
<td>SQL_DIAG_CREATE_ASSERTION</td>
</tr>
<tr>
<td>character-set-definition</td>
<td>&quot;CREATE CHARACTER SET&quot;</td>
<td>SQL_DIAG_CREATE_CHARACTER_SET</td>
</tr>
<tr>
<td>collation-definition</td>
<td>&quot;CREATE COLLATION&quot;</td>
<td>SQL_DIAG_CREATE_COLLATION</td>
</tr>
<tr>
<td>SQL DIAG</td>
<td>SQL DIAG</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
<td></td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td>SQL_DIAG_CREATE_INDEX</td>
<td></td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>SQL_DIAG_CREATE_TABLE</td>
<td></td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>SQL_DIAG_CREATE_VIEW</td>
<td></td>
</tr>
<tr>
<td>SELECT CURSOR</td>
<td>SQL_DIAG_SELECT_CURSOR</td>
<td></td>
</tr>
<tr>
<td>DYNAMIC DELETE CURSOR</td>
<td>SQL_DIAG_DYNAMIC_DELETE_CURSOR</td>
<td></td>
</tr>
<tr>
<td>DELETE WHERE</td>
<td>SQL_DIAG_DELETE WHERE</td>
<td></td>
</tr>
<tr>
<td>CREATE DOMAIN</td>
<td>SQL_DIAG_CREATE_DOMAIN</td>
<td></td>
</tr>
<tr>
<td>DROP ASSERTION</td>
<td>SQL_DIAG_DROP_ASSERTION</td>
<td></td>
</tr>
<tr>
<td>DROP CHARACTER SET</td>
<td>SQL_DIAG_DROP_CHARACTER_SET</td>
<td></td>
</tr>
<tr>
<td>DROP COLLATION</td>
<td>SQL_DIAG_DROP_COLLATION</td>
<td></td>
</tr>
<tr>
<td>DROP DOMAIN</td>
<td>SQL_DIAG_DROP_DOMAIN</td>
<td></td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>SQL_DIAG_DROP_INDEX</td>
<td></td>
</tr>
<tr>
<td>DROP SCHEMA</td>
<td>SQL_DIAG_DROP_SCHEMA</td>
<td></td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>SQL_DIAG_DROP_TABLE</td>
<td></td>
</tr>
<tr>
<td>DROP TRANSLATION</td>
<td>SQL_DIAG_DROP_TRANSLATION</td>
<td></td>
</tr>
</tbody>
</table>
Sequence of Status Records

Status records are positioned in a sequence based on row number and the type of the diagnostic. The Driver Manager determines the final order in which to return status records that it generates. The driver determines the final order in which to return status records that it generates.

If diagnostic records are posted by both the Driver Manager and the driver, the Driver Manager is responsible for ordering them.

If there are two or more status records, the sequence of the records is determined first by row number. The following rules apply to determining the sequence of diagnostic records by row:

- Records that do not correspond to any row appear in front of records that correspond to a particular row, because SQL_NO_ROW_NUMBER is defined to be –1.
- Records for which the row number is unknown appear in front of all other records, because SQL_ROW_NUMBER_UNKNOWN is defined to be –2.
- For all records that pertain to specific rows, records are sorted by the value in the
SQL_DIAG_ROW_NUMBER field. All errors and warnings of the first row affected are listed, and then all errors and warnings of the next row affected, and so on.

**Note**

The ODBC 3.x Driver Manager does not order status records in the diagnostic queue if SQLSTATE 01S01 (Error in row) is returned by an ODBC 2.x driver or if SQLSTATE 01S01 (Error in row) is returned by an ODBC 3.x driver when **SQLExtendedFetch** is called or **SQLSetPos** is called on a cursor that has been positioned with **SQLExtendedFetch**.

Within each row, or for all those records that do not correspond to a row or for which the row number is unknown, or for all those records with a row number equal to SQL_NO_ROW_NUMBER, the first record listed is determined by using a set of sorting rules. After the first record, the order of the other records affecting a row is undefined. An application cannot assume that errors precede warnings after the first record. Applications should scan the complete diagnostic data structure to obtain complete information about an unsuccessful call to a function.

The following rules are used to determine the first record within a row. The record with the highest rank is the first record. The source of a record (Driver Manager, driver, gateway, and so on) is not considered when ranking records.

- **Errors** Status records that describe errors have the highest rank. The following rules are applied to sort errors:
  - Records that indicate a transaction failure or possible transaction failure outrank all other records.
  - If two or more records describe the same error condition, then SQLSTATEs defined by the Open Group CLI specification (classes 03 through HZ) outrank ODBC- and driver-defined SQLSTATEs.

- **Implementation-defined No Data Values** Status records that describe driver-defined No Data values (class 02) have the second highest rank.

- **Warnings** Status records that describe warnings (class 01) have the lowest rank. If two or more records describe the same warning condition, then warning SQLSTATEs defined by the Open Group CLI specification outrank ODBC-defined and driver-defined SQLSTATEs.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtaining multiple fields of a diagnostic data structure</td>
<td><strong>SQLGetDiagRec Function</strong></td>
</tr>
</tbody>
</table>

**See Also**

- ODBC API Reference
- ODBC Header Files
SQLGetDiagRec Function

Conformance
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary
SQLGetDiagRec returns the current values of multiple fields of a diagnostic record that contains error, warning, and status information. Unlike SQLGetDiagField, which returns one diagnostic field per call, SQLGetDiagRec returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the diagnostic message text.

Syntax

```
SQLRETURN SQLGetDiagRec(
    SQLSMALLINT    HandleType,
    SQLHANDLE     Handle,
    SQLSMALLINT    RecNumber,
    SQLCHAR       *    SQLState,
    SQLINTEGER     *    NativeErrorPtr,
    SQLCHAR       *    MessageText,
    SQLSMALLINT    BufferLength,
    SQLSMALLINT   *    TextLengthPtr);
```

Arguments

HandleType
[Input] A handle type identifier that describes the type of handle for which diagnostics are required. Must be one of the following:

- SQL_HANDLE_DBC
- SQL_HANDLE_DBC_INFO_TOKEN
- SQL_HANDLE_DESC
- SQL_HANDLE_ENV
- SQL_HANDLE_STMT

SQL_HANDLE_DBC_INFO_TOKEN handle is used only by the Driver Manager and driver. Applications should not use this handle type. For more information about SQL_HANDLE_DBC_INFO_TOKEN, see Developing Connection-Pool Awareness in an ODBC Driver.
**Handle**

[Input] A handle for the diagnostic data structure, of the type indicated by HandleType. If HandleType is SQL_HANDLE_ENV, Handle can be either a shared or an unshared environment handle.

**RecNumber**

[Input] Indicates the status record from which the application seeks information. Status records are numbered from 1.

**SQLState**

[Output] Pointer to a buffer in which to return a five-character SQLSTATE code (and terminating NULL) for the diagnostic record RecNumber. The first two characters indicate the class; the next three indicate the subclass. This information is contained in the SQL_DIAG_SQLSTATE diagnostic field. For more information, see SQLSTATEs.

**NativeErrorPtr**

[Output] Pointer to a buffer in which to return the native error code, specific to the data source. This information is contained in the SQL_DIAG_NATIVE diagnostic field.

**MessageText**

[Output] Pointer to a buffer in which to return the diagnostic message text string. This information is contained in the SQL_DIAG_MESSAGE_TEXT diagnostic field. For the format of the string, see Diagnostic Messages.

If MessageText is NULL, TextLengthPtr will still return the total number of characters (excluding the null-termination character for character data) available to return in the buffer pointed to by MessageText.

**BufferLength**

[Input] Length of the *MessageText buffer in characters. There is no maximum length of the diagnostic message text.

**TextLengthPtr**

[Output] Pointer to a buffer in which to return the total number of characters (excluding the number of characters required for the null-termination character) available to return in *MessageText. If the number of characters available to return is greater than BufferLength, the diagnostic message text in *MessageText is truncated to BufferLength minus the length of a null-termination character.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

SQLGetDiagRec does not post diagnostic records for itself. It uses the following return values to report the outcome of its own execution:

- SQL_SUCCESS: The function successfully returned diagnostic information.
- SQL_SUCCESS_WITH_INFO: The *MessageText buffer was too small to hold the requested
diagnostic message. No diagnostic records were generated. To determine that a truncation occurred, the application must compare BufferLength to the actual number of bytes available, which is written to *StringLengthPtr.

- SQL_INVALID_HANDLE: The handle indicated by HandleType and Handle was not a valid handle.
- SQL_ERROR: One of the following occurred:
  - RecNumber was negative or 0.
  - BufferLength was less than zero.
  - When using asynchronous notification, the asynchronous operation on the handle was not complete.
- SQL_NO_DATA: RecNumber was greater than the number of diagnostic records that existed for the handle specified in Handle. The function also returns SQL_NO_DATA for any positive RecNumber if there are no diagnostic records for Handle.

Comments

An application typically calls SQLGetDiagRec when a previous call to an ODBC function has returned SQL_ERROR or SQL_SUCCESS_WITH_INFO. However, because any ODBC function can post zero or more diagnostic records each time it is called, an application can call SQLGetDiagRec after any ODBC function call. An application can call SQLGetDiagRec multiple times to return some or all of the records in the diagnostic data structure. ODBC imposes no limit to the number of diagnostic records that can be stored at any one time.

SQLGetDiagRec cannot be used to return fields from the header of the diagnostic data structure. (The RecNumber argument must be greater than 0.) The application should call SQLGetDiagField for this purpose.

SQLGetDiagRec retrieves only the diagnostic information most recently associated with the handle specified in the Handle argument. If the application calls another ODBC function, except SQLGetDiagRec, SQLGetDiagField, or SQLError, any diagnostic information from the previous calls on the same handle is lost.

An application can scan all diagnostic records by looping, incrementing RecNumber, as long as SQLGetDiagRec returns SQL_SUCCESS. Calls to SQLGetDiagRec are nondestructive to the header and record fields. The application can call SQLGetDiagRec again at a later time to retrieve a field from a record as long as no other function, except SQLGetDiagRec, SQLGetDiagField, or SQLError, has been called in the interim. The application can also retrieve a count of the total number of diagnostic records available by calling SQLGetDiagField to retrieve the value of the SQL_DIAG_NUMBER field, and then calling SQLGetDiagRec that many times.

For a description of the fields of the diagnostic data structure, see SQLGetDiagField. For more information, see Using SQLGetDiagRec and SQLGetDiagField and Implementing SQLGetDiagRec and SQLGetDiagField.

Calling an API other than the one that’s being executed asynchronously will generate HY010 "Function sequence error". However, the error record cannot be retrieved before the asynchronous operation completes.
HandleType Argument

Each handle type can have diagnostic information associated with it. The HandleType argument denotes the handle type of the Handle argument.

Some header and record fields cannot be returned for environment, connection, statement, and descriptor handles. Those handles for which a field is not applicable are indicated in the "Header Fields" and "Record Fields" sections in SQLGetDiagField.

A call to SQLGetDiagRec will return SQL_INVALID_HANDLE if HandleType is SQL_HANDLE_SENV, which denotes a shared environment handle. However, if HandleType is SQL_HANDLE_ENV, Handle can be either a shared or an unshared environment handle.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtaining a field of a diagnostic record or a field of the diagnostic header</td>
<td>SQLGetDiagField Function</td>
</tr>
</tbody>
</table>

See Also

- ODBC API Reference
- ODBC Header Files
- Sample ODBC Program

© 2016 Microsoft

SQLGetEnvAttr Function

Conformance

Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary

SQLGetEnvAttr returns the current setting of an environment attribute.

Syntax

```sql
SQLRETURN SQLGetEnvAttr(
    SQLHENV EnvironmentHandle,
    SQLINTEGER Attribute,
)
```
Arguments

EnvironmentHandle
[Input] Environment handle.

Attribute
[Input] Attribute to retrieve.

ValuePtr
[Output] Pointer to a buffer in which to return the current value of the attribute specified by Attribute.

If ValuePtr is NULL, StringLengthPtr will still return the total number of bytes (excluding the null-termination character for character data) available to return in the buffer pointed to by ValuePtr.

BufferLength
[Input] If ValuePtr points to a character string, this argument should be the length of *ValuePtr. If *ValuePtr is an integer, BufferLength is ignored. If *ValuePtr is a Unicode string (when calling SQLGetEnvAttrW), the BufferLength argument must be an even number. If the attribute value is not a character string, BufferLength is unused.

StringLengthPtr
[Output] A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character) available to return in *ValuePtr. If ValuePtr is a null pointer, no length is returned. If the attribute value is a character string and the number of bytes available to return is greater than or equal to BufferLength, the data in *ValuePtr is truncated to BufferLength minus the length of a null-termination character and is null-terminated by the driver.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLGetEnvAttr returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_ENV and a Handle of EnvironmentHandle. The following table lists the SQLSTATE values commonly returned by SQLGetEnvAttr and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The data returned in *ValuePtr was truncated to be BufferLength minus the null-termination character. The length of the untruncated string value is returned in *StringLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) SQL_ATTR_ODBC_VERSION has not yet been set via SQLSetEnvAttr. You do not need to set SQL_ATTR_ODBC_VERSION explicitly if you are using SQLAllocHandleStd.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute&lt;option identifier</td>
<td>The value specified for the argument Attribute was not valid for the version of ODBC supported by the driver.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed. (DM) For more information about suspended state, see SQLEndTran Function.</td>
<td></td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The value specified for the argument Attribute was a valid ODBC environment attribute for the version of ODBC supported by the driver but was not supported by the driver.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver corresponding to the EnvironmentHandle does not support the function.</td>
</tr>
</tbody>
</table>

**Comments**

For a list of attributes, see SQLSetEnvAttr. There are no driver-specific environment attributes. If Attribute specifies an attribute that returns a string, ValuePtr must be a pointer to a buffer in which to return the string. The maximum length of the string, including the null-termination byte, will be BufferLength bytes.
**SQLGetEnvAttr** can be called at any time between the allocation and the freeing of an environment handle. All environment attributes successfully set by the application for the environment persist until **SQLFreeHandle** is called on the **EnvironmentHandle** with a **HandleType** of **SQL_HANDLE_ENV**. More than one environment handle can be allocated simultaneously in ODBC 3.x. An environment attribute on one environment is not affected when another environment has been allocated.

**Note**

The SQL_ATTR_OUTPUT_NTS environment attribute is supported by standards-compliant applications. When **SQLGetEnvAttr** is called, the ODBC 3.x Driver Manager always returns SQL_TRUE for this attribute. SQL_ATTR_OUTPUT_NTS can be set to SQL_TRUE only by a call to **SQLSetEnvAttr**.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returning the setting of a connection attribute</td>
<td><strong>SQLGetConnectAttr Function</strong></td>
</tr>
<tr>
<td>Returning the setting of a statement attribute</td>
<td><strong>SQLGetStmtAttr Function</strong></td>
</tr>
<tr>
<td>Setting a connection attribute</td>
<td><strong>SQLSetConnectAttr Function</strong></td>
</tr>
<tr>
<td>Setting an environment attribute</td>
<td><strong>SQLSetEnvAttr Function</strong></td>
</tr>
<tr>
<td>Setting a statement attribute</td>
<td><strong>SQLSetStmtAttr Function</strong></td>
</tr>
</tbody>
</table>

**See Also**

- [ODBC API Reference](#)
- [ODBC Header Files](#)

© 2016 Microsoft

**SQLGetFunctions Function**

**Conformance**

Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

**Summary**

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

Syntax

```
SQLRETURN SQLGetFunctions(
    SQLHDBC    ConnectionHandle,
    SQLUSMALLINT    FunctionId,
    SQLUSMALLINT *    SupportedPtr);
```

Arguments

**ConnectionHandle**
[Input] Connection handle.

**FunctionId**
[Input] A `#define` value that identifies the ODBC function of interest; `SQL_API_ODBC3_ALL_FUNCTIONS` or `SQL_API_ALL_FUNCTIONS`. `SQL_API_ODBC3_ALL_FUNCTIONS` is used by an ODBC 3.x application to determine support of ODBC 3.x and earlier functions. `SQL_API_ALL_FUNCTIONS` is used by an ODBC 2.x application to determine support of ODBC 2.x and earlier functions.

For a list of `#define` values that identify ODBC functions, see the tables in "Comments."

**SupportedPtr**
[Output] If `FunctionId` identifies a single ODBC function, `SupportedPtr` points to a single SQLUSMALLINT value that is SQL_TRUE if the specified function is supported by the driver, and SQL_FALSE if it is not supported.

If `FunctionId` is `SQL_API_ODBC3_ALL_FUNCTIONS`, `SupportedPtr` points to a SQLUSMALLINT array with a number of elements equal to `SQL_API_ODBC3_ALL_FUNCTIONS_SIZE`. This array is treated by the Driver Manager as a 4,000-bit bitmap that can be used to determine whether an ODBC 3.x or earlier function is supported. The SQL_FUNC_EXISTS macro is called to determine function support. (See "Comments.") An ODBC 3.x application can call `SQLGetFunctions` with `SQL_API_ODBC3_ALL_FUNCTIONS` against either an ODBC 3.x or ODBC 2.x driver.

If `FunctionId` is `SQL_API_ALL_FUNCTIONS`, `SupportedPtr` points to an SQLUSMALLINT array of 100 elements. The array is indexed by `#define` values used by `FunctionId` to identify each ODBC function; some elements of the array are unused and reserved for future use. An element is SQL_TRUE if it identifies an ODBC 2.x or earlier function supported by the driver. It is SQL_FALSE if it identifies an ODBC function not supported by the driver or does not identify an ODBC function.

The arrays returned in `*SupportedPtr` use zero-based indexing.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.
Diagnostics

When SQLGetFunctions returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_DBC and a Handle of ConnectionHandle. The following table lists the SQLSTATE values commonly returned by SQLGetFunctions and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) SQLGetFunctions was called before SQLConnect, SQLBrowseConnect, or SQLDriverConnect.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) SQLBrowseConnect was called for the ConnectionHandle and returned SQL_NEED_DATA. This function was called before SQLBrowseConnect returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the ConnectionHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY095</td>
<td>Function type out of range</td>
<td>(DM) An invalid FunctionId value was specified.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
</tbody>
</table>
The connection timeout period expired before the data source responded to the request. The connection timeout period is set through `SQLSetConnectAttr`, `SQL_ATTR_CONNECTION_TIMEOUT`.

**Comments**

`SQLGetFunctions` always returns that `SQLGetFunctions`, `SQLDataSources`, and `SQLDrivers` are supported. It does this because these functions are implemented in the Driver Manager. The Driver Manager will map an ANSI function to the corresponding Unicode function if the Unicode function exists and will map a Unicode function to the corresponding ANSI function if the ANSI function exists.

For information about how applications use `SQLGetFunctions`, see [Interface Conformance Levels](#).

The following is a list of valid values for `FunctionId` for functions that conform to the ISO 92 standards–compliance level:

<table>
<thead>
<tr>
<th>FunctionId</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_API_SQLALLOCHANDLE</td>
<td>SQL_API_SQLGETDESCFIELD</td>
</tr>
<tr>
<td>SQL_API_SQLBINDCOL</td>
<td>SQL_API_SQLGETDESCREC</td>
</tr>
<tr>
<td>SQL_API_SQLCANCEL</td>
<td>SQL_API_SQLGETDIAGFIELD</td>
</tr>
<tr>
<td>SQL_API_SQLCLOSECURSOR</td>
<td>SQL_API_SQLGETDIAGREC</td>
</tr>
<tr>
<td>SQL_API_SQLCOLATTIBUTE</td>
<td>SQL_API_SQLGETENVATTR</td>
</tr>
<tr>
<td>SQL_API_SQLCONNECT</td>
<td>SQL_API_SQLGETFUNCTIONS</td>
</tr>
<tr>
<td>SQL_API_SQLCOPYDESC</td>
<td>SQL_API_SQLGETINFO</td>
</tr>
<tr>
<td>SQL_API_SQLDATASOURCES</td>
<td>SQL_API_SQLGETSTMTATTR</td>
</tr>
<tr>
<td>SQL_API_SQLDESCRIBECOL</td>
<td>SQL_API_SQLGETTYPEINFO</td>
</tr>
<tr>
<td>SQL_API_SQLDISCONNECT</td>
<td>SQL_API_SQLNUMRESULTCOLS</td>
</tr>
<tr>
<td>SQL_API_SQLDRIVERS</td>
<td>SQL_API_SQLPARAMDATA</td>
</tr>
<tr>
<td>SQL_API_SQLENDTRAN</td>
<td>SQL_API_SQLPREPARE</td>
</tr>
<tr>
<td>SQL_API_SQLEXECDIRECT</td>
<td>SQL_API_SQLPUTDATA</td>
</tr>
<tr>
<td>SQL_API_SQLEXECUTE</td>
<td>SQL_API_SQLROWCOUNT</td>
</tr>
<tr>
<td>SQL_API_SQLFETCH</td>
<td>SQL_API_SQLSETCONNECTATTR</td>
</tr>
</tbody>
</table>
The following is a list of valid values for `FunctionId` for functions conforming to the Open Group standards–compliance level:

<table>
<thead>
<tr>
<th>FunctionId</th>
<th>FunctionId</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_API_SQLCOLUMNS</td>
<td>SQL_API_SQLSTATISTICS</td>
</tr>
<tr>
<td>SQL_API_SQLSPECIALCOLUMNS</td>
<td>SQL_API_SQLTABLES</td>
</tr>
</tbody>
</table>

The following is a list of valid values for `FunctionId` for functions conforming to the ODBC standards–compliance level.

<table>
<thead>
<tr>
<th>FunctionId</th>
<th>FunctionId</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_API_SQLBINDPARAMETER</td>
<td>SQL_API_SQLNATIVESQL</td>
</tr>
<tr>
<td>SQL_API_SQLBROWSECONNECT</td>
<td>SQL_API_SQLNUMPARAMS</td>
</tr>
<tr>
<td>SQL_API_SQLBULKOPERATIONS[1]</td>
<td>SQL_API_SQLPRIMARYKEYS</td>
</tr>
<tr>
<td>SQL_API_SQLCOLUMNPRIVILEGES</td>
<td>SQL_API_SQLPROCEDURECOLUMNS</td>
</tr>
<tr>
<td>SQL_API_SQLDESCRIBEPARAM</td>
<td>SQL_API_SQLPROCEDURES</td>
</tr>
<tr>
<td>SQL_API_SQLDRIVERCONNECT</td>
<td>SQL_API_SQLSETPOS</td>
</tr>
<tr>
<td>SQL_API_SQLFOREIGNKEYS</td>
<td>SQL_API_SQLTABLEPRIVILEGES</td>
</tr>
<tr>
<td>SQL_API_SQLMORERESULTS</td>
<td></td>
</tr>
</tbody>
</table>

[1] When working with an ODBC 2.x driver, `SQLBulkOperations` will be returned as supported only if both of the following are true: the ODBC 2.x driver supports `SQLSetPos`, and the information type `SQL_POS_OPERATIONS` returns the `SQL_POS_ADD` bit as set.

The following is a list of valid values for `FunctionId` for functions introduced in ODBC 3.8 or later:

<table>
<thead>
<tr>
<th>FunctionId</th>
<th>FunctionId</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_API_SQLCANCELHANDLE</td>
<td></td>
</tr>
</tbody>
</table>
SQLCancelHandle will be returned as supported only if the driver supports both SQLCancel and SQLCancelHandle. If SQLCancel is supported but SQLCancelHandle is not, the application can still call SQLCancelHandle on a statement handle, because it will be mapped to SQLCancel.

SQL_FUNC_EXISTS Macro

The SQL_FUNC_EXISTS(SupportedPtr, FunctionID) macro is used to determine support of ODBC 3.x or earlier functions after SQLGetFunctions has been called with a FunctionID argument of SQL_API_ODBC3_ALL_FUNCTIONS. The application calls SQL_FUNC_EXISTS with the SupportedPtr argument set to the SupportedPtr passed in SQLGetFunctions, and with the FunctionID argument set to the #define for the function. SQL_FUNC_EXISTS returns SQL_TRUE if the function is supported, and SQL_FALSE otherwise.

Note

When working with an ODBC 2.x driver, the ODBC 3.x Driver Manager will return SQL_TRUE for SQLAllocHandle and SQLFreeHandle because SQLAllocHandle is mapped to SQLAllocEnv, SQLAllocConnect, or SQLAllocStmt, and because SQLFreeHandle is mapped to SQLFreeEnv, SQLFreeConnect, or SQLFreeStmt. SQLAllocHandle or SQLFreeHandle with a HandleType argument of SQL_HANDLE_DESC is not supported, however, even though SQL_TRUE is returned for the functions, because there is no ODBC 2.x function to map to in this case.

Code Example

The following three examples show how an application uses SQLGetFunctions to determine if a driver supports SQLTables, SQLColumns, and SQLStatistics. If the driver does not support these functions, the application disconnects from the driver. The first example calls SQLGetFunctions once for each function.

In the second example, an ODBC 3.x application calls SQLGetFunctions and passes it an array in
The third example is an ODBC 2.x application calls **SQLGetFunctions** and passes it an array of 100 elements in which **SQLGetFunctions** returns information about all ODBC 2.x and earlier functions.

```
#define FUNCTIONS 100

RETCODE retcodeTables, retcodeColumns, retcodeStatistics
SQLUSMALLINT fExists[FUNCTIONS];

retcode = SQLGetFunctions(hdbc, SQL_API_ALL_FUNCTIONS, fExists);

/* SQLGetFunctions is completed successfully and SQLTables, SQLColumns, and SQLStatistics
if (retcode == SQL_SUCCESS &&
 fExists[SQL_API_SQLTABLES] == SQL_TRUE &&
 fExists[SQL_API_SQLCOLUMNS] == SQL_TRUE &&
 fExists[SQL_API_SQLSTATISTICS] == SQL_TRUE)
{
   /* Continue with application */
}
SQLDisconnect(hdbc);
```

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returning the setting of a connection attribute</td>
<td><strong>SQLGetConnectAttr Function</strong></td>
</tr>
</tbody>
</table>
SQLGetInfo Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLGetInfo returns general information about the driver and data source associated with a connection.

Syntax

```c
SQLRETURN SQLGetInfo(
    SQLHDBC ConnectionHandle,
    SQLUSMALLINT InfoType,
    SQLPOINTER InfoValuePtr,
    SQLSMALLINT BufferLength,
    SQLSMALLINT * StringLengthPtr);
```

Arguments

*ConnectionHandle*
[Input] Connection handle.

*InfoType*
[Input] Type of information.

*InfoValuePtr*
[Output] Pointer to a buffer in which to return the information. Depending on the *InfoType* requested, the information returned will be one of the following: a null-terminated character string, an SQLUSMALLINT value, an SQLUINTEGER bitmask, an SQLUINTEGER flag, a SQLUINTEGER binary value, or a SQLULEN value.
If the `InfoType` argument is SQL_DRIVER_HDESC or SQL_DRIVER_HSTMT, the `InfoValuePtr` argument is both input and output. (See the SQL_DRIVER_HDESC or SQL_DRIVER_HSTMT descriptors later in this function description for more information.)

If `InfoValuePtr` is NULL, `StringLengthPtr` will still return the total number of bytes (excluding the null-termination character for character data) available to return in the buffer pointed to by `InfoValuePtr`.

`BufferLength`

[Input] Length of the `*InfoValuePtr` buffer. If the value in `*InfoValuePtr` is not a character string or if `InfoValuePtr` is a null pointer, the `BufferLength` argument is ignored. The driver assumes that the size of `*InfoValuePtr` is SQLUSMALLINT or SQUINTeger, based on the `InfoType`. If `*InfoValuePtr` is a Unicode string (when calling `SQLGetInfoW`), the `BufferLength` argument must be an even number; if not, SQLSTATE HY090 (Invalid string or buffer length) is returned.

`StringLengthPtr`

[Output] Pointer to a buffer in which to return the total number of bytes (excluding the null-termination character for character data) available to return in `*InfoValuePtr`.

For character data, if the number of bytes available to return is greater than or equal to `BufferLength`, the information in `*InfoValuePtr` is truncated to `BufferLength` bytes minus the length of a null-termination character and is null-terminated by the driver.

For all other types of data, the value of `BufferLength` is ignored and the driver assumes the size of `*InfoValuePtr` is SQLUSMALLINT or SQUINTeger, depending on the `InfoType`.

**Return Value**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When `SQLGetInfo` returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling `SQLGetDiagRec` with a `HandleType` of SQL_HANDLE_DBC and a `Handle` of `ConnectionHandle`. The following table lists the SQLSTATE values typically returned by `SQLGetInfo` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The buffer <code>*InfoValuePtr</code> was not large enough to return all the requested information. Therefore, the information was truncated. The length of the requested information in its untruncated form is returned in <code>*StringLengthPtr</code>. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
<td>(DM) The type of information requested in <code>InfoType</code> requires an open connection. Of the information types reserved by ODBC, only SQL_ODBC_VER can be returned without an open connection.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for the <code>StatementHandle</code> and returned <code>SQL_PARAM_DATA_AVAILABLE</code>. This function was called before data was retrieved for all streamed parameters.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY024</td>
<td>Invalid attribute value</td>
<td>(DM) The <code>InfoType</code> argument was SQL_DRIVER_HSTMT, and the value pointed to by <code>InfoValuePtr</code> was not a valid statement handle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The <code>InfoType</code> argument was SQL_DRIVER_HDESC, and the value pointed to by <code>InfoValuePtr</code> was not a valid descriptor handle.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value specified for argument <code>BufferLength</code> was less than 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The value specified for <code>BufferLength</code> was an odd number, and <code>*InfoValuePtr</code> was of a Unicode data type.</td>
</tr>
<tr>
<td>HY096</td>
<td>Information type out of range</td>
<td>The value specified for the argument <code>InfoType</code> was not valid for the version of ODBC supported by the driver.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see <code>SQLEndTran Function</code>.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional field not implemented</td>
<td>The value specified for the argument <code>InfoType</code> was a driver-specific value that is not supported by the driver.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>, <code>SQL_ATTR_CONNECTION_TIMEOUT</code>.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver that corresponds to the <code>ConnectionHandle</code> does not support the function.</td>
</tr>
</tbody>
</table>

### Comments

The currently defined information types are shown in "Information Types," later in this section; it is expected that more will be defined to take advantage of different data sources. A range of information types is reserved by ODBC; driver developers must reserve values for their own driver-specific use from Open Group. `SQLGetInfo` performs no Unicode conversion or thunking (see Appendix A: ODBC Error Codes of the *ODBC Programmer’s Reference*) for driver-defined `InfoTypes`. For more information, see Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes.

The format of the information returned in `*InfoValuePtr` depends on the `InfoType` requested. `SQLGetInfo` will return information in one of five different formats:

- A null-terminated character string
- An SQLUSMALLINT value
- An SQLUINTEGER bitmask
- An SQLUINTEGER value
- A SQLUINTEGER binary value

The format of each of the following information types is noted in the type's description. The application must cast the value returned in `*InfoValuePtr` accordingly. For an example of how an application could retrieve data from a SQLUINTEGER bitmask, see "Code Example."

A driver must return a value for each information type that is defined in the following tables. If an information type does not apply to the driver or data source, the driver returns one of the values listed in the following table.

#### Character string ("Y" or "N")
- "Y"
- "N"

#### Character string (not "Y" or "N")
- Empty string

#### SQLUSMALLINT
- 0

#### SQLUINTEGER bitmask or SQLUINTEGER binary value
- 0L

For example, if a data source does not support procedures, `SQLGetInfo` returns the values listed in the following table for the values of `InfoType` that are related to procedures.
SQLGetInfo returns SQLSTATE HY096 (Invalid argument value) for values of InfoType that are in the range of information types reserved for use by ODBC but are not defined by the version of ODBC supported by the driver. To determine what version of ODBC a driver complies with, an application calls SQLGetInfo with the SQL_DRIVER_ODBC_VER information type. SQLGetInfo returns SQLSTATE HYC00 (Optional feature not implemented) for values of InfoType that are in the range of information types reserved for driver-specific use but are not supported by the driver.

All calls to SQLGetInfo require an open connection, except when the InfoType is SQL_ODBC_VER, which returns the version of the Driver Manager.

Information Types

This section lists the information types supported by SQLGetInfo. Information types are grouped categorically and listed alphabetically. Information types that were added or renamed for ODBC 3.x are also listed.

Driver Information

The following values of the InfoType argument return information about the ODBC driver, such as the number of active statements, the data source name, and the interface standards compliance level:

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Information Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACTIVE_ENVIRONMENTS</td>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2</td>
</tr>
<tr>
<td>SQL_ASYNCDBC_FUNCTIONS</td>
<td>SQL_FILE_USAGE</td>
</tr>
<tr>
<td>SQL_ASYNC_MODE</td>
<td>SQL_GETDATA_EXTENSIONS</td>
</tr>
<tr>
<td>SQL_ASYNC_NOTIFICATION</td>
<td>SQL_INFO_SCHEMA_VIEWS</td>
</tr>
<tr>
<td>SQL_BATCH_ROW_COUNT</td>
<td>SQL_KEYSET_CURSOR_ATTRIBUTES1</td>
</tr>
<tr>
<td>SQL_BATCH_SUPPORT</td>
<td>SQL_KEYSET_CURSOR_ATTRIBUTES2</td>
</tr>
<tr>
<td>SQL_DATA_SOURCE_NAME</td>
<td>SQL_MAX_ASYNC_CONCURRENT_STATEMENTS</td>
</tr>
<tr>
<td>SQL_DRIVER_AWARE_POOLING_SUPPORTED</td>
<td>SQL_MAX_CONCURRENT_ACTIVITIES</td>
</tr>
</tbody>
</table>
When implementing SQLGetInfo, a driver can improve performance by minimizing the number of times that information is sent or requested from the server.

DBMS Product Information

The following values of the InfoType argument return information about the DBMS product, such as the DBMS name and version:

- SQL_DATABASE_NAME
- SQL_DBMS_NAME
- SQL_DBMS_VER

Data Source Information

The following values of the InfoType argument return information about the data source, such as cursor characteristics and transaction capabilities:

- SQL_ACCESSIBLE_PROCEDURES
- SQL_ACCESSIBLE_TABLES
- SQLBookmarkPersistence
- SQL_CATALOG_TERM
- SQL_MULT_RESULT_SETS
- SQL_MULTIPLE_ACTIVE_TXN
- SQL_NEED_LONG_DATA_LEN
- SQL_NULL_COLLATION
The following values of the *InfoType* argument return information about the SQL statements supported by the data source. The SQL syntax of each feature described by these information types is the SQL-92 syntax. These information types do not exhaustively describe the entire SQL-92 grammar. Instead, they describe those parts of the grammar for which data sources typically offer different levels of support. Specifically, most of the DDL statements in SQL-92 are covered.

Applications should determine the general level of supported grammar from the SQL_SQL_CONFORMANCE information type and use the other information types to determine variations from the stated standards compliance level.

**Supported SQL**

<table>
<thead>
<tr>
<th>SQL_AGGREGATE_FUNCTIONS</th>
<th>SQL_DROP_TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ALTER_DOMAIN</td>
<td>SQL_DROP_TRANSLATION</td>
</tr>
<tr>
<td>SQL_ALTER_SCHEMA</td>
<td>SQL_DROP_VIEW</td>
</tr>
<tr>
<td>SQL_ALTER_TABLE</td>
<td>SQL_EXPRESSIONS_IN_ORDERBY</td>
</tr>
<tr>
<td>SQL_ANSI_SQL_DATETIME_LITERALS</td>
<td>SQL_GROUP_BY</td>
</tr>
<tr>
<td>SQL_CATALOG_LOCATION</td>
<td>SQL_IDENTIFIER_CASE</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME</td>
<td>SQL_IDENTIFIER_QUOTE_CHAR</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME_SEPARATOR</td>
<td>SQL_INDEX_KEYWORDS</td>
</tr>
<tr>
<td>SQL_CATALOG_USAGE</td>
<td>SQL_INSERT_STATEMENT</td>
</tr>
<tr>
<td>SQL_COLUMN_ALIAS</td>
<td>SQL_INTEGRITY</td>
</tr>
<tr>
<td>SQL_CORRELATION_NAME</td>
<td>SQL_KEYWORDS</td>
</tr>
</tbody>
</table>
The following values of the \textit{InfoType} argument return information about the limits applied to identifiers and clauses in SQL statements, such as the maximum lengths of identifiers and the maximum number of columns in a select list. Limitations can be imposed by either the driver or the data source.

<table>
<thead>
<tr>
<th>SQL Limit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_MAX_BINARY_LITERAL_LEN</td>
<td>SQL_MAX_IDENTIFIER_LEN</td>
</tr>
<tr>
<td>SQL_MAX_CATALOG_NAME_LEN</td>
<td>SQL_MAX_INDEX_SIZE</td>
</tr>
<tr>
<td>SQL_MAX_CHAR_LITERAL_LEN</td>
<td>SQL_MAX_PROCEDURE_NAME_LEN</td>
</tr>
<tr>
<td>SQL_MAX_COLUMN_NAME_LEN</td>
<td>SQL_MAX_ROW_SIZE</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_GROUP_BY</td>
<td>SQL_MAX_ROW_SIZE_INCLUDES_LONG</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_INDEX</td>
<td>SQL_MAX_SCHEMA_NAME_LEN</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_ORDER_BY</td>
<td>SQL_MAX_STATEMENT_LEN</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_SELECT</td>
<td>SQL_MAX_TABLE_NAME_LEN</td>
</tr>
</tbody>
</table>

\textbf{SQL Limits}

The following values of the \textit{InfoType} argument return information about the limits applied to identifiers and clauses in SQL statements, such as the maximum lengths of identifiers and the maximum number of columns in a select list. Limitations can be imposed by either the driver or the data source.
Scalar Function Information

The following values of the `InfoType` argument return information about the scalar functions supported by the data source and the driver. For more information about scalar functions, see Appendix E: Scalar Functions.

| SQL_CONVERT_FUNCTIONS                  | SQL_TIMEDATE_ADD_INTERVALS                  |
| SQL_NUMERIC_FUNCTIONS                  | SQL_TIMEDATE_DIFF_INTERVALS                  |
| SQL_STRING_FUNCTIONS                   | SQL_TIMEDATE_FUNCTIONS                      |
| SQL_SYSTEM_FUNCTIONS                   |                                             |

Conversion Information

The following values of the `InfoType` argument return a list of the SQL data types to which the data source can convert the specified SQL data type with the `CONVERT` scalar function:

| SQL_CONVERT_BIGINT                    | SQL_CONVERT_LONGVARBINARY                   |
| SQL_CONVERT_BINARY                    | SQL_CONVERT_LONGVARCHAR                     |
| SQL_CONVERT_BIT                        | SQL_CONVERT_NUMERIC                         |
| SQL_CONVERT_CHAR                        | SQL_CONVERT_REAL                            |
| SQL_CONVERT_DATE                        | SQL_CONVERT_SMALLINT                       |
| SQL_CONVERT_DECIMAL                    | SQL_CONVERT_TIME                            |
| SQL_CONVERT_DOUBLE                     | SQL_CONVERT_TIMESTAMP                       |
| SQL_CONVERT_FLOAT                       | SQL_CONVERT_TINYINT                        |
| SQL_CONVERT_INTEGER                    | SQL_CONVERT_VARBINARY                       |
| SQL_CONVERT_INTERVAL_YEAR_MONTH        | SQL_CONVERT_VARCHAR                        |
| SQL_CONVERT_INTERVAL_DAY_TIME          |                                             |
**Information Types Added for ODBC 3.x**

The following values of the `InfoType` argument have been added for ODBC 3.x:

<table>
<thead>
<tr>
<th>SQL_ACTIVE_ENVIRONMENTS</th>
<th>SQL_DRIVER_AWARE_POOLING_SUPPORTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_AGGREGATE_FUNCTIONS</td>
<td>SQL_DRIVER_HDESC</td>
</tr>
<tr>
<td>SQL_ALTER_DOMAIN</td>
<td>SQL_DROP_ASSERTION</td>
</tr>
<tr>
<td>SQL_ALTER_SCHEMA</td>
<td>SQL_DROP_CHARACTER_SET</td>
</tr>
<tr>
<td>SQL_ANSI_SQL_DATETIME_LITERALS</td>
<td>SQL_DROP_COLLATION</td>
</tr>
<tr>
<td>SQL_ASYNC_DBC_FUNCTIONS</td>
<td>SQL_DROP_DOMAIN</td>
</tr>
<tr>
<td>SQL_ASYNC_MODE</td>
<td>SQL_DROP_SCHEMA</td>
</tr>
<tr>
<td>SQL_ASYNC_NOTIFICATION</td>
<td>SQL_DROP_TABLE</td>
</tr>
<tr>
<td>SQL_BATCH_ROW_COUNT</td>
<td>SQL_DROP_TRANSLATION</td>
</tr>
<tr>
<td>SQL_BATCH_SUPPORT</td>
<td>SQL_DROP_VIEW</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME</td>
<td>SQL_DYNAMIC_CURSOR_ATTRIBUTES1</td>
</tr>
<tr>
<td>SQL_COLLATION_SEQ</td>
<td>SQL_DYNAMIC_CURSOR_ATTRIBUTES2</td>
</tr>
<tr>
<td>SQL_CONVERT_INTERVAL_YEAR_MONTH</td>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1</td>
</tr>
<tr>
<td>SQL_CONVERT_INTERVAL_DAY_TIME</td>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2</td>
</tr>
<tr>
<td>SQL_CREATE_ASSERTION</td>
<td>SQL_INFO_SCHEMA_VIEWS</td>
</tr>
<tr>
<td>SQL_CREATE_CHARACTER_SET</td>
<td>SQL_INSERT_STATEMENT</td>
</tr>
<tr>
<td>SQL_CREATE_COLLATION</td>
<td>SQL_KEYSET_CURSOR_ATTRIBUTES1</td>
</tr>
<tr>
<td>SQL_CREATE_DOMAIN</td>
<td>SQL_KEYSET_CURSOR_ATTRIBUTES2</td>
</tr>
<tr>
<td>SQL_CREATE_SCHEMA</td>
<td>SQL_MAX_ASYNC_CONCURRENT_STATEMENTS</td>
</tr>
<tr>
<td>SQL_CREATE_TABLE</td>
<td>SQL_MAX_IDENTIFIER_LEN</td>
</tr>
<tr>
<td>SQL_CREATE_TRANSLATION</td>
<td>SQL_PARAM_ARRAY_ROW_COUNTS</td>
</tr>
<tr>
<td>SQL_CURSORSENSITIVITY</td>
<td>SQL_PARAM_ARRAY_SELECTS</td>
</tr>
</tbody>
</table>
Information Types Renamed for ODBC 3.x

The following values of the InfoType argument have been renamed for ODBC 3.x.

- SQL_ACTIVE_CONNECTIONS → SQL_MAX_DRIVER_CONNECTIONS
- SQL_ACTIVE_STATEMENTS → SQL_MAX_CONCURRENT_ACTIVITIES
- SQL_MAX_OWNER_NAME_LEN → SQL_MAX_SCHEMA_NAME_LEN
- SQL_MAX_QUALIFIER_NAME_LEN → SQL_MAX_CATALOG_NAME_LEN
- SQL_ODBC_SQL_OPT_IEF → SQL_INTEGRITY
- SQL_OWNER_TERM → SQL_SCHEMA_TERM
- SQL_OWNER_USAGE → SQL_SCHEMA_USAGE
- SQL_QUALIFIER_LOCATION → SQL_CATALOG_LOCATION
- SQL_QUALIFIER_NAME_SEPARATOR → SQL_CATALOG_NAME_SEPARATOR
- SQL_QUALIFIER_TERM → SQL_CATALOG_TERM
- SQL_QUALIFIER_USAGE → SQL_CATALOG_USAGE

Information Types Deprecated in ODBC 3.x

The following values of the InfoType argument have been deprecated in ODBC 3.x. ODBC 3.x drivers must continue to support these information types for backward compatibility with ODBC 2.x applications. (For more information about these types, see SQLGetInfo Support in Appendix G: Driver Guidelines for Backward Compatibility.)
Information Type Descriptions

The following table alphabetically lists each information type, the version of ODBC in which it was introduced, and its description.

<table>
<thead>
<tr>
<th>Information Type Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SQL_ACCESSIBLE_PROCEDURES (ODBC 1.0)</strong></td>
</tr>
<tr>
<td>A character string: &quot;Y&quot; if the user can execute all procedures returned by SQLProcedures; &quot;N&quot; if there may be procedures returned that the user cannot execute.</td>
</tr>
<tr>
<td><strong>SQL_ACCESSIBLE_TABLES (ODBC 1.0)</strong></td>
</tr>
<tr>
<td>A character string: &quot;Y&quot; if the user is guaranteed SELECT privileges to all tables returned by SQLTables; &quot;N&quot; if there may be tables returned that the user cannot access.</td>
</tr>
<tr>
<td><strong>SQL_ACTIVE_ENVIRONMENTS (ODBC 3.0)</strong></td>
</tr>
<tr>
<td>An SQLUSMALLINT value that specifies the maximum number of active environments that the driver can support. If there is no specified limit or the limit is unknown, this value is set to zero.</td>
</tr>
<tr>
<td><strong>SQL_AGGREGATE_FUNCTIONS (ODBC 3.0)</strong></td>
</tr>
<tr>
<td>An SQLUINTEGER bitmask enumerating support for aggregation functions:</td>
</tr>
<tr>
<td>SQL_AF_ALL, SQL_AF_AVG, SQL_AF_COUNT, SQL_AF_DISTINCT, SQL_AF_MAX, SQL_AF_MIN, SQL_AF_SUM</td>
</tr>
<tr>
<td>An SQL-92 Entry level–conformant driver will always return all of these options as supported.</td>
</tr>
<tr>
<td><strong>SQL_ALTER_DOMAIN (ODBC 3.0)</strong></td>
</tr>
<tr>
<td>An SQLUINTEGER bitmask enumerating the clauses in the ALTER DOMAIN statement, as defined in SQL-92, supported by the data source. An SQL-92 Full level–compliant driver will always return all the bitmasks. A return value of &quot;0&quot; means that the ALTER DOMAIN statement is not supported.</td>
</tr>
<tr>
<td>The SQL-92 or FIPS conformance level at which this feature must be supported is shown in parentheses next to each bitmask.</td>
</tr>
<tr>
<td>The following bitmasks are used to determine which clauses are supported:</td>
</tr>
<tr>
<td>SQL_AD_ADD_DOMAIN_CONSTRAINT = Adding a domain constraint is supported (Full level)</td>
</tr>
<tr>
<td>SQL_AD_ADD_DOMAIN_DEFAULT = &lt;alter domain&gt; &lt;set domain default clause&gt; is supported (Full level)</td>
</tr>
<tr>
<td>SQL_AD_CONSTRAINT_NAME_DEFINITION = &lt;constraint name definition clause&gt; is supported for naming domain constraint (Intermediate level)</td>
</tr>
<tr>
<td>SQL_AD_DROP_DOMAIN_CONSTRAINT = &lt;drop domain constraint clause&gt; is supported (Full level)</td>
</tr>
</tbody>
</table>
SQL_AD_DROP_DOMAIN_DEFAULT = <alter domain> <drop domain default clause> is supported (Full level)

The following bits specify the supported <constraint attributes> if <add domain constraint> is supported (the SQL_AD_ADD_DOMAIN_CONSTRAINT bit is set):

SQL_AD_ADD_CONSTRAINT_DEFERRABLE (Full level) SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE (Full level) SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED (Full level) SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE (Full level)

SQL_ALTER_TABLE(ODBC 2.0)
An SQLUINTEGER bitmask enumerating the clauses in the **ALTER TABLE** statement supported by the data source.

The SQL-92 or FIPS conformance level at which this feature must be supported is shown in parentheses next to each bitmask.

The following bitmasks are used to determine which clauses are supported:

SQL_AT_ADD_COLUMN_COLLATION = <add column> clause is supported, with facility to specify column collation (Full level) (ODBC 3.0)

SQL_AT_ADD_COLUMN_DEFAULT = <add column> clause is supported, with facility to specify column defaults (FIPS Transitional level) (ODBC 3.0)

SQL_AT_ADD_COLUMN_SINGLE = <add column> is supported (FIPS Transitional level) (ODBC 3.0)

SQL_AT_ADD_CONSTRAINT = <add column> clause is supported, with facility to specify column constraints (FIPS Transitional level) (ODBC 3.0)

SQL_AT_ADD_TABLE_CONSTRAINT = <add table constraint> clause is supported (FIPS Transitional level) (ODBC 3.0)

SQL_AT_CONSTRAINT_NAME_DEFINITION = <constraint name definition> is supported for naming column and table constraints (Intermediate level) (ODBC 3.0)

SQL_AT_DROP_COLUMN_CASCADE = <drop column> CASCADE is supported (FIPS Transitional level) (ODBC 3.0)

SQL_AT_DROP_COLUMN_DEFAULT = <alter column> <drop column default clause> is supported (Intermediate level) (ODBC 3.0)

SQL_AT_DROP_COLUMN_RESTRICT = <drop column> RESTRICT is supported (FIPS Transitional level) (ODBC 3.0)

SQL_AT_SET_COLUMN_DEFAULT = <alter column> <set column default clause> is supported (Intermediate level) (ODBC 3.0)

The following bits specify the support <constraint attributes> if specifying column or table constraints is supported (the SQL_AT_ADD_CONSTRAINT bit is set):

...
SQL_ASYNC_DBC_FUNCTIONS (ODBC 3.8)
A SQLUINTEGER value that indicates if the driver can execute functions asynchronously on the connection handle.

SQL_ASYNC_DBC_CAPABLE = The driver can execute connection functions asynchronously.

SQL_ASYNC_DBC_NOT_CAPABLE = The driver can not execute connection functions asynchronously.

SQL_ASYNC_MODE(ODBC 3.0)
A SQLUINTEGER value that indicates the level of asynchronous support in the driver:

SQL_AM_CONNECTION = Connection level asynchronous execution is supported. Either all statement handles associated with a given connection handle are in asynchronous mode or all are in synchronous mode. A statement handle on a connection cannot be in asynchronous mode while another statement handle on the same connection is in synchronous mode, and vice versa.

SQL_AM_STATEMENT = Statement level asynchronous execution is supported. Some statement handles associated with a connection handle can be in asynchronous mode, while other statement handles on the same connection are in synchronous mode.

SQL_AM_NONE = Asynchronous mode is not supported.

SQL_ASYNC_NOTIFICATION
A SQLUINTEGER value that indicates if the driver supports asynchronous notification:

- SQL_ASYNC_NOTIFICATION_CAPABLE Asynchronous execution notification is supported by the driver.
- SQL_ASYNC_NOTIFICATION_NOT_CAPABLE Asynchronous execution notification is not supported by the driver.

There are two categories of ODBC asynchronous operations: connection level asynchronous operations and statement level asynchronous operations. If a driver returns SQL_ASYNC_NOTIFICATION_CAPABLE, it must support notification for all APIs that it can execute asynchronously.

SQL_BATCH_ROW_COUNT (ODBC 3.0)
A SQLUINTEGER bitmask that enumerates the behavior of the driver with respect to the availability of row counts. The following bitmasks are used together with the information type:

SQL_BRC_ROLLED_UP = Row counts for consecutive INSERT, DELETE, or UPDATE statements are rolled up into one. If this bit is not set, row counts are available for each statement.

SQL_BRC_PROCEDURES = Row counts, if any, are available when a batch is executed in a stored procedure. If row counts are available, they can be rolled up or individually available, depending on the SQL_BRC_ROLLED_UP bit.

SQL_BRC_EXPLICIT = Row counts, if any, are available when a batch is executed directly by calling SQLExecute or SQLExecDirect. If row counts are available, they can be rolled up or individually available, depending on the SQL_BRC_ROLLED_UP bit.
SQL_BATCH_SUPPORT (ODBC 3.0)

An SQLINTEGER bitmask enumerating the driver’s support for batches. The following bitmasks are used to determine which level is supported:

SQL_BS_SELECT_EXPLICIT = The driver supports explicit batches that can have result-set generating statements.

SQL_BS_ROW_COUNT_EXPLICIT = The driver supports explicit batches that can have row-count generating statements.

SQL_BS_SELECT_PROC = The driver supports explicit procedures that can have result-set generating statements.

SQL_BS_ROW_COUNT_PROC = The driver supports explicit procedures that can have row-count generating statements.

SQL_BOOKMARK_PERSISTENCE(ODBC 2.0)

An SQLINTEGER bitmask enumerating the operations through which bookmarks persist. The following bitmasks are used together with the flag to determine through which options bookmarks persist:

SQL_BP_CLOSE = Bookmarks are valid after an application calls SQLFreeStmt with the SQL_CLOSE option, or SQLCloseCursor to close the cursor associated with a statement.

SQL_BP_DELETE = The bookmark for a row is valid after that row has been deleted.

SQL_BP_DROP = Bookmarks are valid after an application calls SQLFreeHandle with a HandleType of SQL_HANDLE_STMT to drop a statement.

SQL_BP_TRANSACTION = Bookmarks are valid after an application commits or rolls back a transaction.

SQL_BP_UPDATE = The bookmark for a row is valid after any column in that row has been updated, including key columns.

SQL_BP_OTHER_HSTMT = A bookmark associated with one statement can be used with another statement. Unless SQL_BP_CLOSE or SQL_BP_DROP is specified, the cursor on the first statement must be open.

SQL_CATALOG_LOCATION(ODBC 2.0)

An SQLUSMALLINT value that indicates the position of the catalog in a qualified table name:

SQL_CL_STARTSQL_CL_END

For example, an Xbase driver returns SQL_CL_START because the directory (catalog) name is at the start of the table name, as in \EMPDATA\EMP.DBF. An ORACLE Server driver returns SQL_CL_END because the catalog is at the end of the table name, as in ADMIN.EMP@EMPDATA.

An SQL-92 Full level–conformant driver will always return SQL_CL_START. A value of 0 is returned if catalogs are not supported by the data source. To determine whether catalogs are supported, an application calls SQLGetInfo with the SQL_CATALOG_NAME information type.

This InfoType has been renamed for ODBC 3.0 from the ODBC 2.0 InfoType SQL_QUALIFIER_LOCATION.

SQL_CATALOG_NAME(ODBC 3.0)

A character string: "Y" if the server supports catalog names, or "N" if it does not.
An SQL-92 Full level–conformant driver will always return "Y".

**SQL_CATALOG_NAME_SEPARATOR**(ODBC 1.0)
A character string: the character or characters that the data source defines as the separator between a catalog name and the qualified name element that follows or precedes it.

An empty string is returned if catalogs are not supported by the data source. To determine whether catalogs are supported, an application calls `SQLGetInfo` with the `SQL_CATALOG_NAME` information type. An SQL-92 Full level–conformant driver will always return ".".

This *InfoType* has been renamed for ODBC 3.0 from the ODBC 2.0 *InfoType* `SQL_QUALIFIER_NAME_SEPARATOR`.

**SQL_CATALOG_TERM**(ODBC 1.0)
A character string with the data source vendor's name for a catalog; for example, "database" or "directory". This string can be in upper, lower, or mixed case.

An empty string is returned if catalogs are not supported by the data source. To determine whether catalogs are supported, an application calls `SQLGetInfo` with the `SQL_CATALOG_NAME` information type. An SQL-92 Full level–conformant driver will always return "catalog".

This *InfoType* has been renamed for ODBC 3.0 from the ODBC 2.0 *InfoType* `SQL_QUALIFIER_TERM`.

**SQL_CATALOG_USAGE**(ODBC 2.0)
An SQLUINTEGER bitmask enumerating the statements in which catalogs can be used.

The following bitmasks are used to determine where catalogs can be used:

- **SQL_CU_DMLSTATEMENTS** = Catalogs are supported in all Data Manipulation Language statements: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and if supported, `SELECT FOR UPDATE` and positioned update and delete statements.

- **SQL_CU_PROCEDURE_INVOCATION** = Catalogs are supported in the ODBC procedure invocation statement.

- **SQL_CU_TABLE_DEFINITION** = Catalogs are supported in all table definition statements: `CREATE TABLE`, `CREATE VIEW`, `ALTER TABLE`, `DROP TABLE`, and `DROP VIEW`.

- **SQL_CU_INDEX_DEFINITION** = Catalogs are supported in all index definition statements: `CREATE INDEX` and `DROP INDEX`.

- **SQL_CU_PRIVILEGE_DEFINITION** = Catalogs are supported in all privilege definition statements: `GRANT` and `REVOKE`.

A value of 0 is returned if catalogs are not supported by the data source. To determine whether catalogs are supported, an application calls `SQLGetInfo` with the `SQL_CATALOG_NAME` information type. An SQL-92 Full level–conformant driver will always return a bitmask with all of these bits set.

This *InfoType* has been renamed for ODBC 3.0 from the ODBC 2.0 *InfoType* `SQL_QUALIFIER_USAGE`.

**SQL_COLLATION_SEQ**(ODBC 3.0)
The name of the collation sequence. This is a character string that indicates the name of the default collation for the default character set for this server (for example, 'ISO 8859-1' or EBCDIC). If this is unknown, an empty string will be returned. An SQL-92 Full level–conformant driver will always return a non-empty string.
SQL_COLUMN_ALIAS(ODBC 2.0)
A character string: "Y" if the data source supports column aliases; otherwise, "N".

A column alias is an alternative name that can be specified for a column in the select list by using an AS clause. An SQL-92 Entry level–conformant driver will always return "Y".

SQL_CONCAT_NULL_BEHAVIOR(ODBC 1.0)
An SQLUSMALLINT value that indicates how the data source handles the concatenation of NULL valued character data type columns with non-NULL valued character data type columns:

SQL_CB_NULL = Result is NULL valued.
SQL_CB_NON_NULL = Result is concatenation of non-NULL valued column or columns.

An SQL-92 Entry level–conformant driver will always return SQL_CB_NULL.

SQL_CONVERT_FUNCTIONS(ODBC 1.0)
An SQLUINTEGER bitmask enumerating the scalar conversion functions supported by the driver and associated data source.

The following bitmask is used to determine which conversion functions are supported:

SQL_FN_CVT_CASTSQL_FN_CVT_CONVERT

SQL_CORRELATION_NAME(ODBC 1.0)
An SQLUSMALLINT value that indicates whether table correlation names are supported:

SQL_CN_NONE = Correlation names are not supported.
SQL_CN_DIFFERENT = Correlation names are supported but must differ from the names of the tables they represent.
SQL_CN_ANY = Correlation names are supported and can be any valid user-defined name.

An SQL-92 Entry level–conformant driver will always return SQL_CN_ANY.

**SQL_CREATE_ASSERTION(ODBC 3.0)**

An SQLUINTEGER bitmask enumerating the clauses in the CREATE ASSERTION statement, as defined in SQL-92, supported by the data source.

The following bitmasks are used to determine which clauses are supported:

**SQL_CA_CREATE_ASSERTION**

The following bits specify the supported constraint attribute if the ability to specify constraint attributes explicitly is supported (see the SQL_ALTER_TABLE and SQL_CREATE_TABLE information types):

SQL_CA_CONSTRAINT_INITIALLY_DEFERREDSQL_CA_CONSTRAINT_INITIALLY_IMMEDIATESQL_CA_CONSTRAINT_DEFERRABLESQL_CA_CONSTRAINT_NON_DEFERRABLE

An SQL-92 Full level–conformant driver will always return all of these options as supported. A return value of "0" means that the CREATE ASSERTION statement is not supported.

**SQL_CREATE_CHARACTER_SET(ODBC 3.0)**

An SQLUINTEGER bitmask enumerating the clauses in the CREATE CHARACTER SET statement, as defined in SQL-92, supported by the data source.

The following bitmasks are used to determine which clauses are supported:

**SQL_CCS_CREATE_CHARACTER_SETSQL_CCS_COLLATE_CLAUSESQL_CCS_LIMITED_COLLATION**

An SQL-92 Full level–conformant driver will always return all of these options as supported. A return value of "0" means that the CREATE CHARACTER SET statement is not supported.

**SQL_CREATE_COLLATION(ODBC 3.0)**

An SQLUINTEGER bitmask enumerating the clauses in the CREATE COLLATION statement, as defined in SQL-92, supported by the data source.

The following bitmask is used to determine which clauses are supported:

**SQL_CCOL_CREATE_COLLATION**

An SQL-92 Full level–conformant driver will always return this option as supported. A return value of "0" means that the CREATE COLLATION statement is not supported.

**SQL_CREATE_DOMAIN(ODBC 3.0)**

An SQLUINTEGER bitmask enumerating the clauses in the CREATE DOMAIN statement, as defined in SQL-92, supported by the data source.

The following bitmasks are used to determine which clauses are supported:

SQL_CDO_CREATE_DOMAIN = The CREATE DOMAIN statement is supported (Intermediate level).

SQL_CDO_CONSTRAINT_NAME_DEFINITION = <constraint name definition> is supported for naming domain constraints (Intermediate level).

The following bits specify the ability to create column constraints:

SQL_CDO_DEFAULT = Specifying domain constraints is supported (Intermediate level)
SQL_CDO_CONSTRAINT =

Specifying domain defaults is supported (Intermediate level)

SQL_CDO_COLLATION = Specifying domain collation is supported (Full level)

The following bits specify the supported constraint attributes if specifying domain constraints is supported (SQL_CDO_DEFAULT is set):

SQL_CDO.Constraint初始化_DEFERRED (Full level)
SQL_CDO.Constraint初始化_IMMEDIATE (Full level)
SQL_CDO.Constraint_DEFERRABLE (Full level)
SQL_CDO.Constraint_NON_DEFERRABLE (Full level)

A return value of "0" means that the CREATE DOMAIN statement is not supported.

SQL_CREATE_SCHEMA(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses in the CREATE SCHEMA statement, as defined in SQL-92, supported by the data source.

The following bitmasks are used to determine which clauses are supported:

SQL_CS_CREATE_SCHEMA
SQL_CS_AUTHORIZATION
SQL_CS_DEFAULT_CHARACTER_SET

An SQL-92 Intermediate level–conformant driver will always return the SQL_CS_CREATE_SCHEMA and SQL_CS_AUTHORIZATION options as supported. These must also be supported at the SQL-92 Entry level, but not necessarily as SQL statements. An SQL-92 Full level–conformant driver will always return all of these options as supported.

SQL_CREATE_TABLE(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses in the CREATE TABLE statement, as defined in SQL-92, supported by the data source.

The SQL-92 or FIPS conformance level at which this feature must be supported is shown in parentheses next to each bitmask.

The following bitmasks are used to determine which clauses are supported:

SQL_CT_CREATE_TABLE = The CREATE TABLE statement is supported. (Entry level)

SQL_CT_TABLE_CONSTRAINT = Specifying table constraints is supported (FIPS Transitional level)

SQL_CT_CONSTRAINT_NAME_DEFINITION = The <constraint name definition> clause is supported for naming column and table constraints (Intermediate level)

The following bits specify the ability to create temporary tables:

SQL_CT_COMMIT_PRESERVE = Deleted rows are preserved on commit. (Full level)
SQL_CT_COMMIT_DELETE = Deleted rows are deleted on commit. (Full level)
SQL_CT_GLOBAL_TEMPORARY = Global temporary tables can be created. (Full level)
SQL_CT_LOCAL_TEMPORARY = Local temporary tables can be created. (Full level)

The following bits specify the ability to create column constraints:

SQL_CT_COLUMN_CONSTRAINT = Specifying column constraints is supported (FIPS Transitional level)

SQL_CT_COLUMN_DEFAULT = Specifying column defaults is supported (FIPS Transitional level)
SQL_CT_COLUMN_COLLATION = Specifying column collation is supported (Full level)

The following bits specify the supported constraint attributes if specifying column or table constraints is supported:
SQL_CT_CONSTRAINT_INITIALLY_DEFERRED (Full level)
SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE (Full level)
SQL_CT_CONSTRAINT_DEFERRABLE (Full level)
SQL_CT_CONSTRAINT_NON_DEFERRABLE (Full level)

SQL_CREATE_TRANSLATION (ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses in the CREATE TRANSFORMATION statement, as defined in SQL-92, supported by the data source.

The following bitmask is used to determine which clauses are supported:

SQL_CTR_CREATE_TRANSLATION

An SQL-92 Full level-conformant driver will always return these options as supported. A return value of "0" means that the CREATE TRANSFORMATION statement is not supported.

SQL_CREATE_VIEW (ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses in the CREATE VIEW statement, as defined in SQL-92, supported by the data source.

The following bitmasks are used to determine which clauses are supported:

SQL_CV_CREATE_VIEWSQL_CV_CHECK_OPTIONSQL_CV_CASCADESQL_CV_LOCAL

A return value of "0" means that the CREATE VIEW statement is not supported.

An SQL-92 Entry level-conformant driver will always return the SQL_CV_CREATE_VIEW and SQL_CV_CHECK_OPTION options as supported.

An SQL-92 Full level-conformant driver will always return all of these options as supported.

SQL_CURSOR_COMMIT_BEHAVIOR (ODBC 1.0)
An SQLUSMALLINT value that indicates how a COMMIT operation affects cursors and prepared statements in the data source (the behavior of the data source when you commit a transaction).

The value of this attribute will reflect the current state of the next setting:

SQL_COPT_SS_PRESERVE_CURSORS.

SQL_CB_DELETE = Close cursors and delete prepared statements. To use the cursor again, the application must reprepare and reexecute the statement.

SQL_CB_CLOSE = Close cursors. For prepared statements, the application can call SQLExecute on the statement without calling SQLPrepare again. The default for the SQL ODBC driver is SQL_CB_CLOSE. This means that the SQL ODBC driver will close your cursor(s) when you commit a transaction.

SQL_CB_PRESERVE = Preserve cursors in the same position as before the COMMIT operation. The application can continue to fetch data, or it can close the cursor and re-execute the statement without re-preparing it.

SQL_CURSOR_ROLLBACK_BEHAVIOR (ODBC 1.0)
An SQLUSMALLINT value that indicates how a ROLLBACK operation affects cursors and prepared statements in the data source:

SQL_CB_DELETE = Close cursors and delete prepared statements. To use the cursor again, the application must reprepare and reexecute the statement.

SQL_CB_CLOSE = Close cursors. For prepared statements, the application can call SQLExecute on the statement without calling SQLPrepare again.
SQL_CB_PRESERVE = Preserve cursors in the same position as before the **ROLLBACK** operation. The application can continue to fetch data, or it can close the cursor and reexecute the statement without repreparing it.

**SQL_CURR_SSENSTIVITY (ODBC 3.0)**

An SQLUINTEGER value that indicates the support for cursor sensitivity:

- SQL_INSENSITIVE = All cursors on the statement handle show the result set without reflecting any changes that were made to it by any other cursor within the same transaction.

- SQL_UNSPECIFIED = It is unspecified whether cursors on the statement handle make visible the changes that were made to a result set by another cursor within the same transaction. Cursors on the statement handle may make visible none, some, or all such changes.

- SQL_SENSITIVE = Cursors are sensitive to changes that were made by other cursors within the same transaction.

An SQL-92 Entry level–conformant driver will always return the SQL_UNSPECIFIED option as supported.

An SQL-92 Full level–conformant driver will always return the SQL_INSENSITIVE option as supported.

**SQL_DATA_SOURCE_NAME(ODBC 1.0)**

A character string with the data source name that was used during connection. If the application called **SQLConnect**, this is the value of the szDSN argument. If the application called **SQLDriverConnect** or **SQLBrowseConnect**, this is the value of the DSN keyword in the connection string passed to the driver. If the connection string did not contain the **DSN** keyword (such as when it contains the **DRIVER** keyword), this is an empty string.

**SQL_DATA_SOURCE_READ_ONLY(ODBC 1.0)**

A character string. "Y" if the data source is set to READ ONLY mode, "N" if it is otherwise.

This characteristic pertains only to the data source itself; it is not a characteristic of the driver that enables access to the data source. A driver that is read/write can be used with a data source that is read-only. If a driver is read-only, all of its data sources must be read-only and must return SQL_DATA_SOURCE_READ ONLY.

**SQL_DATABASE_NAME(ODBC 1.0)**

A character string with the name of the current database in use, if the data source defines a named object called "database".

**Note**

In ODBC 3.x, the value returned for this **InfoType** can also be returned by calling **SQLGetConnectAttr** with an **Attribute** argument of SQL_ATTR_CURRENT_CATALOG.

**SQL_DATE-TIME_LITERALS(ODBC 3.0)**

An SQLUINTEGER bitmask enumerating the SQL-92 datetime literals supported by the data source. Note that these are the datetime literals listed in the SQL-92 specification and are separate from the datetime literal escape clauses defined by ODBC. For more information about the ODBC datetime literal escape clauses, see [Date, Time, and Timestamp Literals](#).

A FIPS Transitional level–conformant driver will always return the "1" value in the bitmask for the bits in the following list. A value of "0" means that SQL-92 datetime literals are not supported.

The following bitmasks are used to determine which literals are supported:
SQL_DBMS_NAME(ODBC 1.0)
A character string with the name of the DBMS product accessed by the driver.

SQL_DBMS_VER(ODBC 1.0)
A character string that indicates the version of the DBMS product accessed by the driver. The version is of the form ##.##.####, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version. The driver must render the DBMS product version in this form but can also append the DBMS product-specific version. For example, "04.01.0000 Rdb 4.1".

SQL_DDL_INDEX(ODBC 3.0)
An SQLUINTEGER value that indicates support for creation and dropping of indexes:

SQL_DL_CREATE_INDEXSQL_DL_DROP_INDEX

SQL_DEFAULT_TXN_ISOLATION(ODBC 1.0)
An SQLUINTEGER value that indicates the default transaction isolation level supported by the driver or data source, or zero if the data source does not support transactions. The following terms are used to define transaction isolation levels:

**Dirty Read**  Transaction 1 changes a row. Transaction 2 reads the changed row before transaction 1 commits the change. If transaction 1 rolls back the change, transaction 2 will have read a row that is considered to have never existed.

**Nonrepeateable Read** Transaction 1 reads a row. Transaction 2 updates or deletes that row and commits this change. If transaction 1 tries to reread the row, it will receive different row values or discover that the row has been deleted.

**Phantom** Transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 generates one or more rows (through either inserts or updates) that match the search criteria. If transaction 1 reexecutes the statement that reads the rows, it receives a different set of rows.

If the data source supports transactions, the driver returns one of the following bitmasks:

SQL_TXN_READ_UNCOMMITTED = Dirty reads, nonrepeatable reads, and phantoms are possible.

SQL_TXN_READ_COMMITTED = Dirty reads are not possible. Nonrepeatable reads and phantoms are possible.

SQL_TXN_REPEATABLE_READ = Dirty reads and nonrepeatable reads are not possible. Phantoms are possible.

SQL_TXN_SERIALIZABLE = Transactions are serializable. Serializable transactions do not allow dirty reads, nonrepeatable reads, or phantoms.

SQL_DESCRIBE_PARAMETER(ODBC 3.0)
A character string: "Y" if parameters can be described; "N", if not.
An SQL-92 Full level–conformant driver will usually return "Y" because it will support the DESCRIBE INPUT statement. Because this does not directly specify the underlying SQL support, however, describing parameters might not be supported, even in a SQL-92 Full level–conformant driver.

**SQL_DM_VER(ODBC 3.0)**

A character string with the version of the Driver Manager. The version is of the form `##.##.####.#####`, where:

- The first set of two digits is the major ODBC version, as given by the constant SQL_SPEC_MAJOR.
- The second set of two digits is the minor ODBC version, as given by the constant SQL_SPEC_MINOR.
- The third set of four digits is the Driver Manager major build number.
- The last set of four digits is the Driver Manager minor build number.

The Windows 7 Driver Manager version is 03.80. The Windows 8 Driver Manager version is 03.81.

**SQL_DRIVER_AWARE_POOLING_SUPPORTED (ODBC 3.8)**

A SQLUINTEGER value that indicates if the driver support driver-aware pooling. (For more information, see Driver-Aware Connection Pooling.

SQL_DRIVER_AWARE_POOLING_CAPABLE indicates that the driver can support driver-aware pooling mechanism.

SQL_DRIVER_AWARE_POOLING_NOT_CAPABLE indicates that the driver cannot support driver-aware pooling mechanism.

A driver does not need to implement SQL_DRIVER_AWARE_POOLING_SUPPORTED and the Driver Manager will not honor to the driver’s return value.

**SQL_DRIVER_HDBCSQL_DRIVER_HENV(ODBC 1.0)**

An SQLULEN value, the driver's environment handle or connection handle, determined by the argument `InfoType`.

These information types are implemented by the Driver Manager alone.

**SQL_DRIVER_HDESC(ODBC 3.0)**

An SQLULEN value, the driver's descriptor handle determined by the Driver Manager's descriptor handle, which must be passed on input in `*InfoValuePtr` from the application. In this case, `InfoValuePtr` is both an input and output argument. The input descriptor handle passed in `*InfoValuePtr` must have been either explicitly or implicitly allocated on the `ConnectionHandle`.

The application should make a copy of the Driver Manager's descriptor handle before it calls **SQLGetInfo** with this information type, to make sure that the handle is not overwritten on output.

This information type is implemented by the Driver Manager alone.

**SQL_DRIVER_HLIB(ODBC 2.0)**

An SQLULEN value, the `hinst` from the load library returned to the Driver Manager when it loaded the driver DLL on a Microsoft Windows operating system, or its equivalent on another operating system. The handle is valid only for the connection handle specified in the call to SQLGetInfo.
This information type is implemented by the Driver Manager alone.

**SQL_DRIVER_HSTMT (ODBC 1.0)**

An SQLULEN value, the driver's statement handle determined by the Driver Manager statement handle, which must be passed on input in *InfoValuePtr from the application. In this case, *InfoValuePtr is both an input and an output argument. The input statement handle passed in *InfoValuePtr must have been allocated on the argument ConnectionHandle.

The application should make a copy of the Driver Manager's statement handle before it calls SQLGetInfo with this information type, to ensure that the handle is not overwritten on output.

This information type is implemented by the Driver Manager alone.

**SQL_DRIVER_NAME (ODBC 1.0)**

A character string with the file name of the driver used to access the data source.

**SQL_DRIVER_ODBC_VER (ODBC 2.0)**

A character string with the version of ODBC that the driver supports. The version is of the form ##.##, where the first two digits are the major version and the next two digits are the minor version. SQL_SPEC_MAJOR and SQL_SPEC_MINOR define the major and minor version numbers. For the version of ODBC described in this manual, these are 3 and 0, and the driver should return "03.00".

The ODBC Driver Manager will not modify the return value of SQLGetInfo(SQL_DRIVER_ODBC_VER) to maintain backward compatibility for existing applications. The driver specifies which value will be returned. However, a driver that supports C data type extensibility must return 3.8 (or higher) when an application calls SQLSetEnvAttr to set SQL_ATTR_ODBC_VERSION to 3.8. For more information, see C Data Types in ODBC.

**SQL_DRIVER_VER (ODBC 1.0)**

A character string with the version of the driver and optionally, a description of the driver. At a minimum, the version is of the form ##.##.####, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version.

**SQL_DROP_ASSERTION (ODBC 3.0)**

An SQLUINTEGER bitmask enumerating the clauses in the **DROP ASSERTION** statement, as defined in SQL-92, supported by the data source.

The following bitmask is used to determine which clauses are supported:

SQL_DA_DROP_ASSERTION

An SQL-92 Full level-conformant driver will always return this option as supported.

**SQL_DROP_CHARACTER_SET (ODBC 3.0)**

An SQLUINTEGER bitmask enumerating the clauses in the **DROP CHARACTER SET** statement, as defined in SQL-92, supported by the data source.

The following bitmask is used to determine which clauses are supported:

SQL_DCS_DROP_CHARACTER_SET

An SQL-92 Full level-conformant driver will always return this option as supported.

**SQL_DROP_COLLATION (ODBC 3.0)**

An SQLUINTEGER bitmask enumerating the clauses in the **DROP COLLATION** statement, as defined in SQL-92, supported by the data source.
The following bitmask is used to determine which clauses are supported:

SQL_DC_DROP_COLLATION

An SQL-92 Full level-conformant driver will always return this option as supported.

SQL_DROP_DOMAIN(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses in the DROP DOMAIN statement, as defined in SQL-92, supported by the data source.

The following bitmasks are used to determine which clauses are supported:

SQL_DD_DROP_DOMAINSQL_DD_CASCADESQL_DD_RESTRICT

An SQL-92 Intermediate level-conformant driver will always return all of these options as supported.

SQL_DROP_SCHEMA(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses in the DROP SCHEMA statement, as defined in SQL-92, supported by the data source.

The following bitmasks are used to determine which clauses are supported:

SQL_DS_DROP_SCHEMASQL_DS_CASCADESQL_DS_RESTRICT

An SQL-92 Intermediate level-conformant driver will always return all of these options as supported.

SQL_DROP_TABLE(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses in the DROP TABLE statement, as defined in SQL-92, supported by the data source.

The following bitmasks are used to determine which clauses are supported:

SQL_DT_DROP_TABLESQL_DT_CASCADESQL_DT_RESTRICT

An FIPS Transitional level-conformant driver will always return all of these options as supported.

SQL_DROP_TRANSLATION(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses in the DROP TRANSLATION statement, as defined in SQL-92, supported by the data source.

The following bitmask is used to determine which clauses are supported:

SQL_DTR_DROP_TRANSLATION

An SQL-92 Full level-conformant driver will always return this option as supported.

SQL_DROP_VIEW(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses in the DROP VIEW statement, as defined in SQL-92, supported by the data source.

The following bitmasks are used to determine which clauses are supported:

SQL_DV_DROP_VIEWSQL_DV_CASCADESQL_DV_RESTRICT

An FIPS Transitional level-conformant driver will always return all of these options as supported.

SQL_DYNAMIC_CURSOR_ATTRIBUTES1(ODBC 3.0)
An SQLUINTEGER bitmask that describes the attributes of a dynamic cursor that are supported
by the driver. This bitmask contains the first subset of attributes; for the second subset, see 
SQL_DYNAMIC_CURSOR_ATTRIBUTES2.

The following bitmasks are used to determine which attributes are supported:

SQL_CA1_NEXT = A FetchOrientation argument of SQL_FETCH_NEXT is supported in a call to 
SQLFetchScroll when the cursor is a dynamic cursor.

SQL_CA1_ABSOLUTE = FetchOrientation arguments of SQL_FETCH_FIRST, SQL_FETCH_LAST, 
and SQL_FETCH_ABSOLUTE are supported in a call to SQLFetchScroll when the cursor is a 
dynamic cursor. (The rowset that will be fetched is independent of the current cursor position.)

SQL_CA1_RELATIVE = FetchOrientation arguments of SQL_FETCH_PRIOR and 
SQL_FETCH_RELATIVE are supported in a call to SQLFetchScroll when the cursor is a dynamic 
cursor. (The rowset that will be fetched depends on the current cursor position. Note that this is 
separated from SQL_FETCH_NEXT because in a forward-only cursor, only SQL_FETCH_NEXT is 
supported.)

SQL_CA1_BOOKMARK = A FetchOrientation argument of SQL_FETCH_BOOKMARK is supported 
in a call to SQLFetchScroll when the cursor is a dynamic cursor.

SQL_CA1_LOCK_EXCLUSIVE = A LockType argument of SQL_LOCK_EXCLUSIVE is supported in 
a call to SQLSetPos when the cursor is a dynamic cursor.

SQL_CA1_LOCK_NO_CHANGE = A LockType argument of SQL_LOCK_NO_CHANGE is supported 
in a call to SQLSetPos when the cursor is a dynamic cursor.

SQL_CA1_LOCK_UNLOCK = A LockType argument of SQL_LOCK_UNLOCK is supported in a call 
to SQLSetPos when the cursor is a dynamic cursor.

SQL_CA1_POS_POSITION = An Operation argument of SQL_POSITION is supported in a call to 
SQLSetPos when the cursor is a dynamic cursor.

SQL_CA1_POS_UPDATE = An Operation argument of SQL_UPDATE is supported in a call to 
SQLSetPos when the cursor is a dynamic cursor.

SQL_CA1_POS_DELETE = An Operation argument of SQL_DELETE is supported in a call to 
SQLSetPos when the cursor is a dynamic cursor.

SQL_CA1_POS_REFRESH = An Operation argument of SQL_REFRESH is supported in a call to 
SQLSetPos when the cursor is a dynamic cursor.

SQL_CA1_POSITIONED_UPDATE = An UPDATE WHERE CURRENT OF SQL statement is supported 
when the cursor is a dynamic cursor. (An SQL-92 Entry level–conformant driver will always 
return this option as supported.)

SQL_CA1_POSITIONED_DELETE = A DELETE WHERE CURRENT OF SQL statement is supported 
when the cursor is a dynamic cursor. (An SQL-92 Entry level–conformant driver will always 
return this option as supported.)

SQL_CA1_SELECT_FOR_UPDATE = A SELECT FOR UPDATE SQL statement is supported when 
the cursor is a dynamic cursor. (An SQL-92 Entry level–conformant driver will always return this 
option as supported.)

SQL_CA1_BULK_ADD = An Operation argument of SQL_ADD is supported in a call to 
SQLBulkOperations when the cursor is a dynamic cursor.

SQL_CA1_BULK_UPDATE_BY_BOOKMARK = An Operation argument of 
SQL_UPDATE_BY_BOOKMARK is supported in a call to SQLBulkOperations when the cursor is
A dynamic cursor.

SQL_CA1_BULK_DELETE_BY_BOOKMARK = An Operation argument of SQL_DELETE_BY_BOOKMARK is supported in a call to SQLBulkOperations when the cursor is a dynamic cursor.

SQL_CA1_BULK_FETCH_BY_BOOKMARK = An Operation argument of SQL_FETCH_BY_BOOKMARK is supported in a call to SQLBulkOperations when the cursor is a dynamic cursor.

An SQL-92 Intermediate level–conformant driver will usually return the SQL_CA1_NEXT, SQL_CA1_ABSOLUTE, and SQL_CA1_RELATIVE options as supported, because it supports scrollable cursors through the embedded SQL FETCH statement. Because this does not directly determine the underlying SQL support, however, scrollable cursors may not be supported, even for an SQL-92 Intermediate level–conformant driver.

SQL_DYNAMIC_CURSOR_ATTRIBUTES2(ODBC 3.0)
An SQLUINTEGER bitmask that describes the attributes of a dynamic cursor that are supported by the driver. This bitmask contains the second subset of attributes; for the first subset, see SQL_DYNAMIC_CURSOR_ATTRIBUTES1.

The following bitmasks are used to determine which attributes are supported:

SQL_CA2_READ_ONLY_CONCURRENCY = A read-only dynamic cursor, in which no updates are allowed, is supported. (The SQL_ATTR_CONCURRENCY statement attribute can be SQL_CONCUR_READ_ONLY for a dynamic cursor).

SQL_CA2_LOCK_CONCURRENCY = A dynamic cursor that uses the lowest level of locking sufficient to make sure that the row can be updated is supported. (The SQL_ATTR_CONCURRENCY statement attribute can be SQL_CONCUR_LOCK for a dynamic cursor.) These locks must be consistent with the transaction isolation level set by the SQL_ATTR_TXN_ISOLATION connection attribute.

SQL_CA2_OPT_ROWVER_CONCURRENCY = A dynamic cursor that uses the optimistic concurrency control comparing row versions is supported. (The SQL_ATTR_CONCURRENCY statement attribute can be SQL_CONCUR_ROWVER for a dynamic cursor.)

SQL_CA2_OPT_VALUES_CONCURRENCY = A dynamic cursor that uses the optimistic concurrency control comparing values is supported. (The SQL_ATTR_CONCURRENCY statement attribute can be SQL_CONCUR_VALUES for a dynamic cursor.)

SQL_CA2_SENSITIVITY_ADDITIONS = Added rows are visible to a dynamic cursor; the cursor can scroll to those rows. (Where these rows are added to the cursor is driver-dependent.)

SQL_CA2_SENSITIVITY_DELETIONS = Deleted rows are no longer available to a dynamic cursor, and do not leave a "hole" in the result set; after the dynamic cursor scrolls from a deleted row, it cannot return to that row.

SQL_CA2_SENSITIVITY_UPDATES = Updates to rows are visible to a dynamic cursor; if the dynamic cursor scrolls from and returns to an updated row, the data returned by the cursor is the updated data, not the original data.

SQL_CA2_MAX_ROWS_SELECT = The SQL_ATTR_MAX_ROWS statement attribute affects SELECT statements when the cursor is a dynamic cursor.

SQL_CA2_MAX_ROWS_INSERT = The SQL_ATTR_MAX_ROWS statement attribute affects INSERT statements when the cursor is a dynamic cursor.
SQL_CA2_MAX_ROWS_DELETE = The SQL_ATTR_MAX_ROWS statement attribute affects DELETE statements when the cursor is a dynamic cursor.

SQL_CA2_MAX_ROWS_UPDATE = The SQL_ATTR_MAX_ROWS statement attribute affects UPDATE statements when the cursor is a dynamic cursor.

SQL_CA2_MAX_ROWS_CATALOG = The SQL_ATTR_MAX_ROWS statement attribute affects CATALOG result sets when the cursor is a dynamic cursor.

SQL_CA2_MAX_ROWS_AFFECTS_ALL = The SQL_ATTR_MAX_ROWS statement attribute affects SELECT, INSERT, DELETE, and UPDATE statements, and CATALOG result sets, when the cursor is a dynamic cursor.

SQL_CA2_CRC_EXACT = The exact row count is available in the SQL_DIAG_CURSOR_ROW_COUNT diagnostic field when the cursor is a dynamic cursor.

SQL_CA2_CRC_APPROXIMATE = An approximate row count is available in the SQL_DIAG_CURSOR_ROW_COUNT diagnostic field when the cursor is a dynamic cursor.

SQL_CA2_SIMULATE_NON_UNIQUE = The driver does not guarantee that simulated positioned update or delete statements will affect only one row when the cursor is a dynamic cursor; it is the application's responsibility to guarantee this. (If a statement affects more than one row, SQLExecute or SQLExecDirect returns SQLSTATE 01001 [Cursor operation conflict].) To set this behavior, the application calls SQLSetStmtAttr with the SQL_ATTR_SIMULATE_CURSOR attribute set to SQL_SC_NON_UNIQUE.

SQL_CA2_SIMULATE_TRY_UNIQUE = The driver tries to guarantee that simulated positioned update or delete statements will affect only one row when the cursor is a dynamic cursor. The driver always executes such statements, even if they might affect more than one row, such as when there is no unique key. (If a statement affects more than one row, SQLExecute or SQLExecDirect returns SQLSTATE 01001 [Cursor operation conflict].) To set this behavior, the application calls SQLSetStmtAttr with the SQL_ATTR_SIMULATE_CURSOR attribute set to SQL_SC_TRY_UNIQUE.

SQL_CA2_SIMULATE_UNIQUE = The driver guarantees that simulated positioned update or delete statements will affect only one row when the cursor is a dynamic cursor. If the driver cannot guarantee this for a given statement, SQLExecDirect or SQLPrepare return SQLSTATE 01001 (Cursor operation conflict). To set this behavior, the application calls SQLSetStmtAttr with the SQL_ATTR_SIMULATE_CURSOR attribute set to SQL_SC_UNIQUE.

SQL_EXPRESSIONS_IN_ORDERBY(ODBC 1.0)
A character string: "Y" if the data source supports expressions in the ORDER BY list; "N" if it does not.

SQL_FILE_USAGE(ODBC 2.0)
An SQLUSMALLINT value that indicates how a single-tier driver directly treats files in a data source:

SQL_FILE_NOT_SUPPORTED = The driver is not a single-tier driver. For example, an ORACLE driver is a two-tier driver.

SQL_FILE_TABLE = A single-tier driver treats files in a data source as tables. For example, an Xbase driver treats each Xbase file as a table.

SQL_FILE_CATALOG = A single-tier driver treats files in a data source as a catalog. For example, a Microsoft Access driver treats each Microsoft Access file as a complete database.

An application might use this to determine how users will select data. For example, Xbase users
often think of data as stored in files, whereas ORACLE and Microsoft Access users generally think of data as stored in tables.

When a user selects an Xbase data source, the application could display the Windows File Open common dialog box; when the user selects a Microsoft Access or ORACLE data source, the application could display a custom Select Table dialog box.

**SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1(ODBC 3.0)**

An SQLINTEGER bitmask that describes the attributes of a forward-only cursor that are supported by the driver. This bitmask contains the first subset of attributes; for the second subset, see SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2.

The following bitmasks are used to determine which attributes are supported:

```
SQL_CA1_NEXTSQL_CA1_LOCK_EXCLUSIVESQL_CA1_LOCK_NO_CHANGESQL_CA1_LOCK_UNLOCKSQL_CA1_POS_POSITIONSQL_CA1_POS_UPDATESQL_CA1_POS_DELETESQL_CA1_POS_REFRESHSQL_CA1_BULK_ADDSQL_CA1_BULK_UPDATE_BY_BOOKMARKSQL_CA1_BULK_DELETE _BY_BOOKMARKSQL_CA1_BULK_FETCH_BY_BOOKMARK
```

For descriptions of these bitmasks, see SQL_DYNAMIC_CURSOR_ATTRIBUTES1 (and substitute "forward-only cursor" for "dynamic cursor" in the descriptions).

**SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2(ODBC 3.0)**

An SQLINTEGER bitmask that describes the attributes of a forward-only cursor that are supported by the driver. This bitmask contains the second subset of attributes; for the first subset, see SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1.

The following bitmasks are used to determine which attributes are supported:

```
SQL_CA2_READ_ONLY_CONCURRENCYSQL_CA2_LOCK_CONCURRENCYSQL_CA2_OPT_ROWVER CONCURRENCYSQL_CA2_OPT_VALUES_CONCURRENCYSQL_CA2_SENSITIVITY_ADDITIONSSQ L_CA2_SENSITIVITY_DELETIONS_SQL_CA2_SENSITIVITY_UPDATESSQL_CA2_MAX_ROWS_SELECTSQL_CA2_MAX_ROWS_INSERTSQL_CA2_MAX_ROWS_DELETESSQL_CA2_MAX_ROWS_UPDATESQL_CA2_MAX_ROWS_CATALOGSQL_CA2_MAX_ROWS_AFFECTS_ALLSQL_CA2_CRC_EXACTSQL_CA2_CRC_APPROXIMATESQL_CA2_SIMULATE_NON_UNIQUESQL_CA2_SIMULATE_TRY_UNIQUE
```

For descriptions of these bitmasks, see SQL_DYNAMIC_CURSOR_ATTRIBUTES2 (and substitute "forward-only cursor" for "dynamic cursor" in the descriptions).

**SQL_GETDATA_EXTENSIONS(ODBC 2.0)**

An SQLINTEGER bitmask enumerating extensions to SQLGetData.

The following bitmasks are used together with the flag to determine what common extensions the driver supports for SQLGetData:

```
SQL_GD_ANY_COLUMN = SQLGetData can be called for any unbound column, including those before the last bound column. Note that the columns must be called in order of ascending column number unless SQL_GD_ANY_ORDER is also returned.

SQL_GD_ANY_ORDER = SQLGetData can be called for unbound columns in any order. Note that SQLGetData can be called only for columns after the last bound column unless SQL_GD_ANY_COLUMN is also returned.

SQL_GD_BLOCK = SQLGetData can be called for an unbound column in any row in a block (where the rowset size is greater than 1) of data after positioning to that row with SQLSetPos.

SQL_GD_BOUND = SQLGetData can be called for bound columns in addition to unbound
columns. A driver cannot return this value unless it also returns SQL_GD_ANY_COLUMN.

SQL_GD_OUTPUT_PARAMS = SQLGetData can be called to return output parameter values. For more information, see Retrieving Output Parameters Using SQLGetData.

SQLGetData is required to return data only from unbound columns that occur after the last bound column, are called in order of increasing column number, and are not in a row in a block of rows.

If a driver supports bookmarks (either fixed-length or variable-length), it must support calling SQLGetData on column 0. This support is required regardless of what the driver returns for a call to SQLGetInfo with the SQL_GETDATA_EXTENSIONS InfoType.

SQL_GROUP_BY(ODBC 2.0)
An SQLUSMALLINT value that specifies the relationship between the columns in the GROUP BY clause and the nonaggregated columns in the select list:

SQL_GB_COLLATE = A COLLATE clause can be specified at the end of each grouping column. (ODBC 3.0)

SQL_GB_NOT_SUPPORTED = GROUP BY clauses are not supported. (ODBC 2.0)

SQL_GB_GROUP_BY_EQUALS_SELECT = The GROUP BY clause must contain all nonaggregated columns in the select list. It cannot contain any other columns. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT. (ODBC 2.0)

SQL_GB_GROUP_BY_CONTAINS_SELECT = The GROUP BY clause must contain all nonaggregated columns in the select list. It can contain columns that are not in the select list. For example, SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT, AGE. (ODBC 2.0)

SQL_GB_NO_RELATION = The columns in the GROUP BY clause and the select list are not related. The meaning of nongrouped, nonaggregated columns in the select list is data source–dependent. For example, SELECT DEPT, SALARY FROM EMPLOYEE GROUP BY DEPT, AGE. (ODBC 2.0)

An SQL-92 Entry level–conformant driver will always return the SQL_GB_GROUP_BY_EQUALS_SELECT option as supported. An SQL-92 Full level–conformant driver will always return the SQL_GB_COLLATE option as supported. If none of the options is supported, the GROUP BY clause is not supported by the data source.

SQL_IDENTIFIER_CASE(ODBC 1.0)
An SQLUSMALLINT value as follows:

SQL_IC_UPPER = Identifiers in SQL are not case-sensitive and are stored in uppercase in system catalog.

SQL_IC_LOWER = Identifiers in SQL are not case-sensitive and are stored in lowercase in system catalog.

SQL_IC_SENSITIVE = Identifiers in SQL are case sensitive and are stored in mixed case in system catalog.

SQL_IC_MIXED = Identifiers in SQL are not case-sensitive and are stored in mixed case in system catalog.

Because identifiers in SQL-92 are never case-sensitive, a driver that conforms strictly to SQL-92 (any level) will never return the SQL_IC_SENSITIVE option as supported.
SQL_IDENTIFIER_QUOTE_CHAR(ODBC 1.0)

The character string that is used as the starting and ending delimiter of a quoted (delimited) identifier in SQL statements. (Identifiers passed as arguments to ODBC functions do not have to be quoted.) If the data source does not support quoted identifiers, a blank is returned.

This character string can also be used for quoting catalog function arguments when the connection attribute SQL_ATTR_METADATA_ID is set to SQL_TRUE.

Because the identifier quote character in SQL-92 is the double quotation mark ("), a driver that conforms strictly to SQL-92 will always return the double quotation mark character.

SQL_INDEX_KEYWORDS(ODBC 3.0)

An SQLINTEGER bitmask that enumerates keywords in the CREATE INDEX statement that are supported by the driver:

- SQL_IK_NONE = None of the keywords is supported.
- SQL_IK_ASC = ASC keyword is supported.
- SQL_IK_DESC = DESC keyword is supported.
- SQL_IK_ALL = All keywords are supported.

To see whether the CREATE INDEX statement is supported, an application calls SQLGetInfo with the SQL_DLL_INDEX information type.

SQL_INFO_SCHEMA_VIEWS(ODBC 3.0)

An SQLINTEGER bitmask enumerating the views in the INFORMATION_SCHEMA that are supported by the driver. The views in, and the contents of, INFORMATION_SCHEMA are as defined in SQL-92.

The SQL-92 or FIPS conformance level at which this feature must be supported is shown in parentheses next to each bitmask.

The following bitmasks are used to determine which views are supported:

- SQL_ISV_ASSERTIONS = Identifies the catalog's assertions that are owned by a given user. (Full level)
- SQL_ISV_CHARACTER_SETS = Identifies the catalog's character sets that can be accessed by a given user. (Intermediate level)
- SQL_ISV_CHECK_CONSTRAINTS = Identifies the CHECK constraints that are owned by a given user. (Intermediate level)
- SQL_ISV_COLLATIONS = Identifies the character collations for the catalog that can be accessed by a given user. (Full level)
- SQL_ISV_COLUMN_DOMAIN_USAGE = Identifies columns for the catalog that depend on domains defined in the catalog and are owned by a given user. (Intermediate level)
- SQL_ISV_COLUMN_PRIVILEGES = Identifies the privileges on columns of persistent tables that are available to or granted by a given user. (FIPS Transitional level)
- SQL_ISV_COLUMNS = Identifies the columns of persistent tables that can be accessed by a given user. (FIPS Transitional level)
- SQL_ISV_CONSTRAINT_COLUMN_USAGE = Similar to CONSTRAINT_TABLE_USAGE view, columns are identified for the various constraints that are owned by a given user. (Intermediate
SQL_ISV_CONSTRAINT_TABLE_USAGE = Identifies the tables that are used by constraints (referential, unique, and assertions), and are owned by a given user. (Intermediate level)

SQL_ISV_DOMAIN_CONSTRAINTS = Identifies the domain constraints (of the domains in the catalog) that can be accessed by a given user. (Intermediate level)

SQL_ISV_DOMAINS = Identifies the domains defined in a catalog that can be accessed by the user. (Intermediate level)

SQL_ISV_KEY_COLUMN_USAGE = Identifies columns defined in the catalog that are constrained as keys by a given user. (Intermediate level)

SQL_ISV_REFERENTIAL_CONSTRAINTS = Identifies the referential constraints that are owned by a given user. (Intermediate level)

SQL_ISV_SCHEMATA = Identifies the schemas that are owned by a given user. (Intermediate level)

SQL_ISV_SQL_LANGUAGES = Identifies the SQL conformance levels, options, and dialects supported by the SQL implementation. (Intermediate level)

SQL_ISV_TABLE_CONSTRAINTS = Identifies the table constraints that are owned by a given user. (Intermediate level)

SQL_ISV_TABLE_PRIVILEGES = Identifies the privileges on persistent tables that are available to or granted by a given user. (FIPS Transitional level)

SQL_ISV_TABLES = Identifies the persistent tables defined in a catalog that can be accessed by a given user. (FIPS Transitional level)

SQL_ISV_TRANSLATIONS = Identifies character translations for the catalog that can be accessed by a given user. (Full level)

SQL_ISV_USAGE_PRIVILEGES = Identifies the USAGE privileges on catalog objects that are available to or owned by a given user. (FIPS Transitional level)

SQL_ISV_VIEW_COLUMN_USAGE = Identifies the columns on which the catalog's views that are owned by a given user are dependent. (Intermediate level)

SQL_ISV_VIEW_TABLE_USAGE = Identifies the tables on which the catalog's views that are owned by a given user are dependent. (Intermediate level)

SQL_ISV_VIEWS = Identifies the viewed tables defined in this catalog that can be accessed by a given user. (FIPS Transitional level)

SQL_INSERT_STATEMENT(ODBC 3.0)
An SQLINTEGER bitmask that indicates support for INSERT statements:

SQL_IS_INSERT_LITERALS
SQL_IS_INSERT_SEARCHED
SQL_IS_SELECT_INTO

An SQL-92 Entry level–conformant driver will always return all of these options as supported.

SQL_INTEGRITY(ODBC 1.0)
A character string: "Y" if the data source supports the Integrity Enhancement Facility; "N" if it
does not.

This *InfoType* has been renamed for ODBC 3.0 from the ODBC 2.0 *InfoType* SQL_ODBC_SQL_OPT_IEF.

**SQL_KEYSET_CURSOR_ATTRIBUTES1(ODBC 3.0)**

An SQLUINTEGER bitmask that describes the attributes of a keyset cursor that are supported by the driver. This bitmask contains the first subset of attributes; for the second subset, see SQL_KEYSET_CURSOR_ATTRIBUTES2.

The following bitmasks are used to determine which attributes are supported:

SQL_CA1_NEXTSQL_CA1_ABSOLUTESQL_CA1_RELATIVESQL_CA1_BOOKMARKSQL_CA1_LOCK _EXCLUSIVESQL_CA1_LOCK_NO_CHANGESQL_CA1_LOCK_UNLOCKSQL_CA1_POSPOSITIONSQL _CA1_POS_UPDATESQL_CA1_POS_DELETESQL_CA1_POS_REFRESHSQL_CA1_POSITIONED_UP DATESQL_CA1_POSITIONED_DELETESQL_CA1_SELECT_FOR_UPDATESQL_CA1_BULK_ADDSQL_ CA1_BULK_UPDATE_BY_BOOKMARKSQL_CA1_BULK_DELETE_BY_BOOKMARKSQL_CA1_BULK_F ETCH_BY_BOOKMARK

For descriptions of these bitmasks, see SQL_DYNAMIC_CURSOR_ATTRIBUTES1 (and substitute "keyset-driven cursor" for "dynamic cursor" in the descriptions).

An SQL-92 Intermediate level–conformant driver will usually return the SQL_CA1_NEXT, SQL_CA1_ABSOLUTE, and SQL_CA1_RELATIVE options as supported, because the driver supports scrollable cursors through the embedded SQL FETCH statement. Because this does not directly determine the underlying SQL support, however, scrollable cursors may not be supported, even for an SQL-92 Intermediate level–conformant driver.

**SQL_KEYSET_CURSOR_ATTRIBUTES2(ODBC 3.0)**

An SQLUINTEGER bitmask that describes the attributes of a keyset cursor that are supported by the driver. This bitmask contains the second subset of attributes; for the first subset, see SQL_KEYSET_CURSOR_ATTRIBUTES1.

The following bitmasks are used to determine which attributes are supported:

SQL_CA2_READ_ONLY_CONCURRENCYSQL_CA2_LOCK_CONCURRENCYSQL_CA2_OPT_ROWVER _CONCURRENCYSQL_CA2_OPT_VALUES_CONCURRENCYSQL_CA2_SENSITIVITY_ADDITIONSSQ L_CA2_SENSITIVITY_DELETIONSQL_CA2_SENSITIVITY_UPDATESSQL_CA2_MAX_ROWS_SELECTSQL_CA2_MAX_ROWS_INSERTSQL_CA2_MAX_ROWS_DELETESQL_CA2_MAX_ROWS_UPDATE SQL_CA2_MAX_ROWS_CATALOGSQL_CA2_MAX_ROWS_AFFECTS_ALLSQL_CA2_CRC_EXACTSQL _CA2_CRC_APPROXIMATESQL_CA2_SIMULATE_NON_EQUIVALENTSQL_CA2_SIMULATE_TRY_UNIQUE SQL_CA2_SIMULATE_UNIQUE

For descriptions of these bitmasks, see SQL_DYNAMIC_CURSOR_ATTRIBUTES1 (and substitute "keyset-driven cursor" for "dynamic cursor" in the descriptions).

**SQL_KEYWORDS(ODBC 2.0)**

A character string that contains a comma-separated list of all data source–specific keywords. This list does not contain keywords specific to ODBC or keywords used by both the data source and ODBC. This list represents all the reserved keywords; interoperable applications should not use these words in object names.

For a list of ODBC keywords, see [Reserved Keywords in Appendix C: SQL Grammar](#). The `#define` value SQL_ODBC_KEYWORDS contains a comma-separated list of ODBC keywords.

Appendix C: SQL Grammar

**SQL_LIKE_ESCAPE_CLAUSE(ODBC 2.0)**
A character string: "Y" if the data source supports an escape character for the percent character (%) and underscore character (_) in a LIKE predicate and the driver supports the ODBC syntax for defining a LIKE predicate escape character; "N" otherwise.

**SQL_MAX_ASYNC_CONCURRENT_STATEMENTS(ODBC 3.0)**
An SQLINTEGER value that specifies the maximum number of active concurrent statements in asynchronous mode that the driver can support on a given connection. If there is no specific limit or the limit is unknown, this value is zero.

**SQL_MAX_BINARY_LITERAL_LEN(ODBC 2.0)**
An SQLINTEGER value that specifies the maximum length (number of hexadecimal characters, excluding the literal prefix and suffix returned by SQLGetTypeInfo) of a binary literal in an SQL statement. For example, the binary literal 0xFFAA has a length of 4. If there is no maximum length or the length is unknown, this value is set to zero.

**SQL_MAX_CATALOG_NAME_LEN(ODBC 1.0)**
An SQLUSMALLINT value that specifies the maximum length of a catalog name in the data source. If there is no maximum length or the length is unknown, this value is set to zero.

An FIPS Full level–conformant driver will return at least 128.

This InfoType has been renamed for ODBC 3.0 from the ODBC 2.0 InfoType SQL_MAX_QUALIFIER_NAME_LEN.

**SQL_MAX_CHAR_LITERAL_LEN(ODBC 2.0)**
An SQLINTEGER value that specifies the maximum length (number of characters, excluding the literal prefix and suffix returned by SQLGetTypeInfo) of a character literal in an SQL statement. If there is no maximum length or the length is unknown, this value is set to zero.

**SQL_MAX_COLUMN_NAME_LEN(ODBC 1.0)**
An SQLUSMALLINT value that specifies the maximum length of a column name in the data source. If there is no maximum length or the length is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 18. An FIPS Intermediate level–conformant driver will return at least 128.

**SQL_MAX_COLUMNS_IN_GROUP_BY(ODBC 2.0)**
An SQLUSMALLINT value that specifies the maximum number of columns allowed in a GROUP BY clause. If there is no specified limit or the limit is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 6. An FIPS Intermediate level–conformant driver will return at least 15.

**SQL_MAX_COLUMNS_IN_INDEX(ODBC 2.0)**
An SQLUSMALLINT value that specifies the maximum number of columns allowed in an index. If there is no specified limit or the limit is unknown, this value is set to zero.

**SQL_MAX_COLUMNS_IN_ORDER_BY(ODBC 2.0)**
An SQLUSMALLINT value that specifies the maximum number of columns allowed in an ORDER BY clause. If there is no specified limit or the limit is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 6. An FIPS Intermediate level–conformant driver will return at least 15.

**SQL_MAX_COLUMNS_IN_SELECT(ODBC 2.0)**
An SQLUSMALLINT value that specifies the maximum number of columns allowed in a select list. If there is no specified limit or the limit is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 100. An FIPS Intermediate level–
conformant driver will return at least 250.

**SQL_MAX_COLUMNS_IN_TABLE (ODBC 2.0)**

An SQLSMALLINT value that specifies the maximum number of columns allowed in a table. If there is no specified limit or the limit is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 100. An FIPS Intermediate level–conformant driver will return at least 250.

**SQL_MAX_CONCURRENT_ACTIVITIES (ODBC 1.0)**

An SQLSMALLINT value that specifies the maximum number of active statements that the driver can support for a connection. A statement is defined as active if it has results pending, with the term "results" meaning rows from a **SELECT** operation or rows affected by an **INSERT**, **UPDATE**, or **DELETE** operation (such as a row count), or if it is in a NEED_DATA state. This value can reflect a limitation imposed by either the driver or the data source. If there is no specified limit or the limit is unknown, this value is set to zero.

This *InfoType* has been renamed for ODBC 3.0 from the ODBC 2.0 *InfoType* SQL_ACTIVE_STATEMENTS.

**SQL_MAX_CURSOR_NAME_LEN (ODBC 1.0)**

An SQLSMALLINT value that specifies the maximum length of a cursor name in the data source. If there is no maximum length or the length is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 18. An FIPS Intermediate level–conformant driver will return at least 128.

**SQL_MAX_DRIVER_CONNECTIONS (ODBC 1.0)**

An SQLSMALLINT value that specifies the maximum number of active connections that the driver can support for an environment. This value can reflect a limitation imposed by either the driver or the data source. If there is no specified limit or the limit is unknown, this value is set to zero.

This *InfoType* has been renamed for ODBC 3.0 from the ODBC 2.0 *InfoType* SQL_ACTIVE_CONNECTIONS.

**SQL_MAX_IDENTIFIER_LEN (ODBC 3.0)**

An SQLSMALLINT that indicates the maximum size in characters that the data source supports for user-defined names.

An FIPS Entry level–conformant driver will return at least 18. An FIPS Intermediate level–conformant driver will return at least 128.

**SQL_MAX_INDEX_SIZE (ODBC 2.0)**

An SQLINTEGER value that specifies the maximum number of bytes allowed in the combined fields of an index. If there is no specified limit or the limit is unknown, this value is set to zero.

**SQL_MAX_PROCEDURE_NAME_LEN (ODBC 1.0)**

An SQLSMALLINT value that specifies the maximum length of a procedure name in the data source. If there is no maximum length or the length is unknown, this value is set to zero.

**SQL_MAX_ROW_SIZE (ODBC 2.0)**

An SQLINTEGER value that specifies the maximum length of a single row in a table. If there is no specified limit or the limit is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 2,000. An FIPS Intermediate level–conformant driver will return at least 8,000.

**SQL_MAX_ROW_SIZE_INCLUDES_LONG (ODBC 3.0)**
A character string: "Y" if the maximum row size returned for the SQL_MAX_ROW_SIZE information type includes the length of all SQL_LONGVARCHAR and SQL_LONGVARBINARY columns in the row; "N" otherwise.

**SQL_MAX_SCHEMA_NAME_LEN(ODBC 1.0)**

An SQLUSMALLINT value that specifies the maximum length of a schema name in the data source. If there is no maximum length or the length is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 18. An FIPS Intermediate level–conformant driver will return at least 128.

This *InfoType* has been renamed for ODBC 3.0 from the ODBC 2.0 *InfoType* SQL_MAX_OWNER_NAME_LEN.

**SQL_MAX_STATEMENT_LEN(ODBC 2.0)**

An SQLUINTEGER value that specifies the maximum length (number of characters, including white space) of an SQL statement. If there is no maximum length or the length is unknown, this value is set to zero.

**SQL_MAX_TABLE_NAME_LEN(ODBC 1.0)**

An SQLUSMALLINT value that specifies the maximum length of a table name in the data source. If there is no maximum length or the length is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 18. An FIPS Intermediate level–conformant driver will return at least 128.

**SQL_MAX_TABLES_IN_SELECT(ODBC 2.0)**

An SQLUSMALLINT value that specifies the maximum number of tables allowed in the **FROM** clause of a **SELECT** statement. If there is no specified limit or the limit is unknown, this value is set to zero.

An FIPS Entry level–conformant driver will return at least 15. An FIPS Intermediate level–conformant driver will return at least 50.

**SQL_MAX_USER_NAME_LEN(ODBC 2.0)**

An SQLUSMALLINT value that specifies the maximum length of a user name in the data source. If there is no maximum length or the length is unknown, this value is set to zero.

**SQL_MULT_RESULT_SETS(ODBC 1.0)**

A character string: "Y" if the data source supports multiple result sets, "N" if it does not.

For more information about multiple result sets, see Multiple Results.

**SQL_MULTIPLE_ACTIVE_TXN(ODBC 1.0)**

A character string: "Y" if the driver supports more than one active transaction at the same time, "N" if only one transaction can be active at any time.

The information returned for this information type does not apply in the case of distributed transactions.

**SQL_NEED_LONG_DATA_LEN(ODBC 2.0)**

A character string: "Y" if the data source needs the length of a long data value (the data type is SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long data source–specific data type) before that value is sent to the data source, "N" if it does not. For more information, see SQLBindParameter Function and SQLSetPos Function.

**SQL_NON_NULLABLE_COLUMNS(ODBC 1.0)**

An SQLUSMALLINT value that specifies whether the data source supports NOT NULL in columns:
SQL_NNC_NULL = All columns must be nullable.

SQL_NNC_NON_NULL = Columns cannot be nullable. (The data source supports the NOT NULL column constraint in CREATE TABLE statements.)

An SQL-92 Entry level–conformant driver will return SQL_NNC_NON_NULL.

SQL_NULL_COLLATION(ODBC 2.0)
An SQLUSMALLINT value that specifies where NULLs are sorted in a result set:

SQL_NC_END = NULLs are sorted at the end of the result set, regardless of the ASC or DESC keywords.

SQL_NC_HIGH = NULLs are sorted at the high end of the result set, depending on the ASC or DESC keywords.

SQL_NC_LOW = NULLs are sorted at the low end of the result set, depending on the ASC or DESC keywords.

SQL_NC_START = NULLs are sorted at the start of the result set, regardless of the ASC or DESC keywords.

SQL_NUMERIC_FUNCTIONS(ODBC 1.0)
Note: The information type was introduced in ODBC 1.0; each bitmask is labeled with the version in which it was introduced.

An SQLUINTEGER bitmask enumerating the scalar numeric functions supported by the driver and associated data source.

The following bitmasks are used to determine which numeric functions are supported:

SQL_FN_NUM_ABS (ODBC 1.0) SQL_FN_NUM_ACOS (ODBC 1.0) SQL_FN_NUM_ASIN (ODBC 1.0) SQL_FN_NUM_ATAN (ODBC 1.0) SQL_FN_NUM_ATAN2 (ODBC 1.0) SQL_FN_NUM_CEILING (ODBC 1.0) SQL_FN_NUM_COS (ODBC 1.0) SQL_FN_NUM_COT (ODBC 1.0) SQL_FN_NUM_DEGREES (ODBC 2.0) SQL_FN_NUM_EXP (ODBC 1.0) SQL_FN_NUM_FLOOR (ODBC 1.0) SQL_FN_NUM_LOG (ODBC 1.0) SQL_FN_NUM_LOG10 (ODBC 2.0) SQL_FN_NUM_MOD (ODBC 1.0) SQL_FN_NUM_PI (ODBC 1.0) SQL_FN_NUM_POWER (ODBC 2.0) SQL_FN_NUM_RADIANS (ODBC 2.0) SQL_FN_NUM_RAND (ODBC 1.0) SQL_FN_NUM_ROUND (ODBC 2.0) SQL_FN_NUM_SIGN (ODBC 1.0) SQL_FN_NUM_SIN (ODBC 1.0) SQL_FN_NUM_SQRT (ODBC 1.0) SQL_FN_NUM_TRUNCATE (ODBC 2.0)

SQL_ODBC_INTERFACE_CONFORMANCE(ODBC 3.0)
An SQLUINTEGER value that indicates the level of the ODBC 3.x interface that the driver complies with.

SQL_OIC_CORE: The minimum level that all ODBC drivers are expected to comply with. This level includes basic interface elements such as connection functions, functions for preparing and executing an SQL statement, basic result set metadata functions, basic catalog functions, and so on.

SQL_OIC_LEVEL1: A level including the core standards compliance level functionality, plus scrollable cursors, bookmarks, positioned updates and deletes, and so on.

SQL_OIC_LEVEL2: A level including level 1 standards compliance level functionality, plus advanced features such as sensitive cursors; update, delete, and refresh by bookmarks; stored procedure support; catalog functions for primary and foreign keys; multi-catalog support; and so on.

For more information, see Interface Conformance Levels.
SQL_ODBC_VER(ODBC 1.0)
A character string with the version of ODBC to which the Driver Manager conforms. The version is of the form ##.##.##.0000, where the first two digits are the major version and the next two digits are the minor version. This is implemented only in the Driver Manager.

SQL_OJ_CAPABILITIES(ODBC 2.01)
An SQLUINTEGER bitmask enumerating the types of outer joins supported by the driver and data source. The following bitmasks are used to determine which types are supported:

- SQL_OJ_LEFT = Left outer joins are supported.
- SQL_OJ_RIGHT = Right outer joins are supported.
- SQL_OJ_FULL = Full outer joins are supported.
- SQL_OJ_NESTED = Nested outer joins are supported.
- SQL_OJ_NOT_ORDERED = The column names in the ON clause of the outer join do not have to be in the same order as their respective table names in the OUTER JOIN clause.
- SQL_OJ_INNER = The inner table (the right table in a left outer join or the left table in a right outer join) can also be used in an inner join. This does not apply to full outer joins, which do not have an inner table.
- SQL_OJ_ALL_COMPARISON_OPS = The comparison operator in the ON clause can be any of the ODBC comparison operators. If this bit is not set, only the equals (=) comparison operator can be used in outer joins.

If none of these options is returned as supported, no outer join clause is supported.

For information about the support of relational join operators in a SELECT statement, as defined by SQL-92, see SQL_SQL92_RELATIONAL_JOIN_OPERATORS.

SQL_ORDER_BY_COLUMNS_IN_SELECT(ODBC 2.0)
A character string: "Y" if the columns in the ORDER BY clause must be in the select list; otherwise, "N".

SQL_PARAM_ARRAY_ROW_COUNTS(ODBC 3.0)
An SQLUINTEGER enumerating the driver's properties regarding the availability of row counts in a parameterized execution. Has the following values:

- SQL_PARC_BATCH = Individual row counts are available for each set of parameters. This is conceptually equivalent to the driver generating a batch of SQL statements, one for each parameter set in the array. Extended error information can be retrieved by using the SQL_PARAM_STATUS_PTR descriptor field.
- SQL_PARC_NO_BATCH = There is only one row count available, which is the cumulative row count resulting from the execution of the statement for the entire array of parameters. This is conceptually equivalent to treating the statement together with the complete parameter array as one atomic unit. Errors are handled the same as if one statement were executed.

SQL_PARAM_ARRAY_SELECTS(ODBC 3.0)
An SQLUINTEGER enumerating the driver's properties regarding the availability of result sets in a parameterized execution. Has the following values:

- SQL_PAS_BATCH = There is one result set available per set of parameters. This is conceptually equivalent to the driver generating a batch of SQL statements, one for each parameter set in the array.
SQL_PAS_NO_BATCH = There is only one result set available, which represents the cumulative result set resulting from the execution of the statement for the complete array of parameters. This is conceptually equivalent to treating the statement together with the complete parameter array as one atomic unit.

SQL_PAS_NO_SELECT = A driver does not allow a result-set generating statement to be executed with an array of parameters.

SQL_PROCEDURE_TERM(ODBC 1.0)
A character string with the data source vendor's name for a procedure; for example, "database procedure", "stored procedure", "procedure", "package", or "stored query".

SQL_PROCEDURES(ODBC 1.0)
A character string: "Y" if the data source supports procedures and the driver supports the ODBC procedure invocation syntax; "N" otherwise.

SQL_POS_OPERATIONS(ODBC 2.0)
An SQLINTEGER bitmask enumerating the support operations in SQLSetPos.

The following bitmasks are used together with the flag to determine which options are supported.

SQL_POS_POSITION (ODBC 2.0) SQL_POS_REFRESH (ODBC 2.0) SQL_POS_UPDATE (ODBC 2.0) SQL_POS_DELETE (ODBC 2.0) SQL_POS_ADD (ODBC 2.0)

SQL_QUOTED_IDENTIFIER_CASE(ODBC 2.0)
An SQLUSMALLINT value as follows:

SQL_IC_UPPER = Quoted identifiers in SQL are not case-sensitive and are stored in uppercase in the system catalog.

SQL_IC_LOWER = Quoted identifiers in SQL are not case-sensitive and are stored in lowercase in the system catalog.

SQL_IC_SENSITIVE = Quoted identifiers in SQL are case sensitive and are stored in mixed case in the system catalog. (In an SQL-92–compliant database, quoted identifiers are always case-sensitive.)

SQL_IC_MIXED = Quoted identifiers in SQL are not case-sensitive and are stored in mixed case in the system catalog.

An SQL-92 Entry level–conformant driver will always return SQL_IC_SENSITIVE.

SQL_ROW_UPDATES(ODBC 1.0)
A character string: "Y" if a keyset-driven or mixed cursor maintains row versions or values for all fetched rows and therefore can detect any updates that were made to a row by any user since the row was last fetched. (This applies only to updates, not to deletions or insertions.) The driver can return the SQL_ROW_UPDATED flag to the row status array when SQLFetchScroll is called. Otherwise, "N".

SQL_SCHEMA_TERM(ODBC 1.0)
A character string with the data source vendor's name for a schema; for example, "owner", "Authorization ID", or "Schema".

The character string can be returned in upper, lower, or mixed case.

An SQL-92 Entry level–conformant driver will always return "schema".

This InfoType has been renamed for ODBC 3.0 from the ODBC 2.0 InfoType
**SQL_SCHEMA_USAGE (ODBC 2.0)**

An SQLUINTEGER bitmask enumerating the statements in which schemas can be used:

- **SQL_SU_DML_STATEMENTS** = Schemas are supported in all Data Manipulation Language statements: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and if supported, `SELECT FOR UPDATE` and positioned update and delete statements.

- **SQL_SU_PROCEDURE_INVOCATION** = Schemas are supported in the ODBC procedure invocation statement.

- **SQL_SU_TABLE_DEFINITION** = Schemas are supported in all table definition statements: `CREATE TABLE`, `CREATE VIEW`, `ALTER TABLE`, `DROP TABLE`, and `DROP VIEW`.

- **SQL_SU_INDEX_DEFINITION** = Schemas are supported in all index definition statements: `CREATE INDEX` and `DROP INDEX`.

- **SQL_SU_PRIVILEGE_DEFINITION** = Schemas are supported in all privilege definition statements: `GRANT` and `REVOKE`.

An SQL-92 Entry level–conformant driver will always return the **SQL_SU_DML_STATEMENTS**, **SQL_SU_TABLE_DEFINITION**, and **SQL_SU_PRIVILEGE_DEFINITION** options, as supported.

This *InfoType* has been renamed for ODBC 3.0 from the ODBC 2.0 *InfoType* SQL_OWNER_USAGE.

**SQL_SCROLL_OPTIONS (ODBC 1.0)**

Note: The information type was introduced in ODBC 1.0; each bitmask is labeled with the version in which it was introduced.

An SQLUINTEGER bitmask enumerating the scroll options supported for scrollable cursors.

The following bitmasks are used to determine which options are supported:

- **SQL_SO_FORWARD_ONLY** = The cursor only scrolls forward. (ODBC 1.0)

- **SQL_SO_STATIC** = The data in the result set is static. (ODBC 2.0)

- **SQL_SO_KEYSET_DRIVEN** = The driver saves and uses the keys for every row in the result set. (ODBC 1.0)

- **SQL_SO_DYNAMIC** = The driver keeps the keys for every row in the rowset (the keyset size is the same as the rowset size). (ODBC 1.0)

- **SQL_SO_MIXED** = The driver keeps the keys for every row in the keyset, and the keyset size is greater than the rowset size. The cursor is keyset-driven inside the keyset and dynamic outside the keyset. (ODBC 1.0)

For information about scrollable cursors, see *Scrollable Cursors*.

**SQL_SEARCH_PATTERN_ESCAPE (ODBC 1.0)**

A character string specifying what the driver supports as an escape character that allows the use of the pattern match metacharacters underscore (_) and percent sign (%) as valid characters in search patterns. This escape character applies only for those catalog function arguments that support search strings. If this string is empty, the driver does not support a search-pattern escape character.

Because this information type does not indicate general support of the escape character in the
LIKE predicate, SQL-92 does not include requirements for this character string.

This InfoType is limited to catalog functions. For a description of the use of the escape character in search pattern strings, see Pattern Value Arguments.

SQL_SERVER_NAME(ODBC 1.0)
A character string with the actual data source–specific server name; useful when a data source name is used during SQLConnect, SQLDriverConnect, and SQLBrowseConnect.

SQL_SPECIAL_CHARACTERS(ODBC 2.0)
A character string that contains all special characters (that is, all characters except a through z, A through Z, 0 through 9, and underscore) that can be used in an identifier name, such as a table name, column name, or index name, on the data source. For example, "#$^". If an identifier contains one or more of these characters, the identifier must be a delimited identifier.

SQL_SQL_CONFORMANCE(ODBC 3.0)
An SQLUINTEGER value that indicates the level of SQL-92 supported by the driver:

SQL_SC_SQL92_ENTRY = Entry level SQL-92 compliant.
SQL_SC_FIPS127_2_TRANSITIONAL = FIPS 127-2 transitional level compliant.
SQL_SC_SQL92_FULL = Full level SQL-92 compliant.
SQL_SC_SQL92_INTERMEDIATE = Intermediate level SQL-92 compliant.

SQL_SQL92_DATETIME_FUNCTIONS(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the datetime scalar functions that are supported by the driver and the associated data source, as defined in SQL-92.

The following bitmasks are used to determine which datetime functions are supported:

SQL_SDF_CURRENT_DATESQL_SDF_CURRENT_TIMESQL_SDF_CURRENT_TIMESTAMPSQL_SDF_CURRENT_DATE

SQL_SQL92_FOREIGN_KEY_DELETE_RULE(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the rules supported for a foreign key in a DELETE statement, as defined in SQL-92.

The following bitmasks are used to determine which clauses are supported by the data source:

SQL_SFKD_CASCADESQL_SFKD_NO_ACTIONSQL_SFKD_SET_DEFAULTSQL_SFKD_SET_NULL

An FIPS Transitional level–conformant driver will always return all of these options as supported.

SQL_SQL92_FOREIGN_KEY_UPDATE_RULE(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the rules supported for a foreign key in an UPDATE statement, as defined in SQL-92.

The following bitmasks are used to determine which clauses are supported by the data source:

SQL_SFKU_CASCADESQL_SFKU_NO_ACTIONSQL_SFKU_SET_DEFAULTSQL_SFKU_SET_NULL

An SQL-92 Full level–conformant driver will always return all of these options as supported.

SQL_SQL92_GRANT(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the clauses supported in the GRANT statement, as defined in SQL-92.

The SQL-92 or FIPS conformance level at which this feature must be supported is shown in parentheses next to each bitmask.
The following bitmasks are used to determine which clauses are supported by the data source:

- SQL_SG_DELETE_TABLE (Entry level)
- SQL_SG_INSERT_COLUMN (Intermediate level)
- SQL_SG_INSERT_TABLE (Entry level)
- SQL_SG_REFERENCES_TABLE (Entry level)
- SQL_SG_REFERENCES_COLUMN (Entry level)
- SQL_SG_SELECT_TABLE (Entry level)
- SQL_SG_UPDATE_COLUMN (Entry level)
- SQL_SG_UPDATE_TABLE (Entry level)
- SQL_SG_USAGE_ON_DOMAIN (FIPS Transitional level)
- SQL_SG_USAGE_ON_CHARACTER_SET (FIPS Transitional level)
- SQL_SG_USAGE_ON_COLLATION (FIPS Transitional level)
- SQL_SG_USAGE_ON_TRANSLATION (FIPS Transitional level)
- SQL_SG_WITH_GRANT_OPTION (Entry level)

An SQLUINTEGER bitmask enumerating the numeric value scalar functions that are supported by the driver and the associated data source, as defined in SQL-92.

The following bitmasks are used to determine which numeric functions are supported:

- SQL_SNVF_BIT_LENGTH
- SQL_SNVF_CHAR_LENGTH
- SQL_SNVF_CHARACTER_LENGTH
- SQL_SNVF_EXTRACT
- SQL_SNVF_OCTET_LENGTH
- SQL_SNVF_POSITION

An SQLUINTEGER bitmask enumerating the predicates supported in a SELECT statement, as defined in SQL-92.

The following bitmasks are used to determine which options are supported by the data source:

- SQL_SP_BETWEEN (Entry level)
- SQL_SP_COMPARISON (Entry level)
- SQL_SP_EXISTS (Entry level)
- SQL_SP_IN (Entry level)
- SQL_SP_ISNOTNULL (Entry level)
- SQL_SP_ISNULL (Entry level)
- SQL_SP_LIKE (Entry level)
- SQL_SP_MATCH_FULL (Full level)
- SQL_SP_MATCH_PARTIAL (Full level)
- SQL_SP_MATCH_UNIQUE_FULL (Full level)
- SQL_SP_MATCH_UNIQUE_PARTIAL (Full level)
- SQL_SP_OVERLAPS (FIPS Transitional level)
- SQL_SP_QUANTIFIED_COMPARISON (Entry level)
- SQL_SP_UNIQUE (Entry level)

An SQLUINTEGER bitmask enumerating the relational join operators supported in a SELECT statement, as defined in SQL-92.

The following bitmasks are used to determine which options are supported by the data source:

- SQL_SRJO_CORRESPONDING_CLAUSE (Intermediate level)
- SQL_SRJO_CROSS_JOIN (Full level)
- SQL_SRJO_EXCEPT_JOIN (Intermediate level)
- SQL_SRJO_FULLSCREEN_OUTER_JOIN (Intermediate level)
- SQL_SRJO_INNER_JOIN (FIPS Transitional level)
- SQL_SRJO_INTERSECT_JOIN (Intermediate level)
- SQL_SRJO_LEFT_OUTER_JOIN (FIPS Transitional level)
- SQL_SRJO_NATURAL_JOIN (FIPS Transitional level)
- SQL_SRJO_RIGHT_OUTER_JOIN (FIPS Transitional level)
- SQL_SRJO_UNION_JOIN (Full level)

SQL_SRJO_INNER_JOIN indicates support for the INNER JOIN syntax, not for the inner join capability. Support for the INNER JOIN syntax is FIPS TRANSITIONAL, whereas support for the inner join capability is ENTRY.

An SQLUINTEGER bitmask enumerating the clauses supported in the REVOKE statement, as
defined in SQL-92, supported by the data source.

The SQL-92 or FIPS conformance level at which this feature must be supported is shown in parentheses next to each bitmask.

The following bitmasks are used to determine which clauses are supported by the data source:

SQL_SR_CASCADE (FIPS Transitional level) SQL_SR_DELETE_TABLE (Entry level) SQL_SR_GRANT_OPTION_FOR (Intermediate level) SQL_SR_INSERT_COLUMN (Intermediate level) SQL_SR_INSERT_TABLE (Entry level) SQL_SR_REFERENCES_COLUMN (Entry level) SQL_SR_REFERENCES_TABLE (Entry level) SQL_SR_RESTRICT (FIPS Transitional level) SQL_SR_SELECT_TABLE (Entry level) SQL_SR_UPDATE_COLUMN (Entry level) SQL_SR_UPDATE_TABLE (Entry level) SQL_SR_USAGE_ON_DOMAIN (FIPS Transitional level) SQL_SR_USAGE_ON_CHARACTER_SET (FIPS Transitional level) SQL_SR_USAGE_ON_COLLATION (FIPS Transitional level) SQL_SR_USAGE_ON_TRANSLATION (FIPS Transitional level)

SQL_SQL92_ROW_VALUE_CONSTRUCTOR(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the row value constructor expressions supported in a SELECT statement, as defined in SQL-92. The following bitmasks are used to determine which options are supported by the data source:

SQL_SRVC_VALUE_EXPRESSION SQL_SRVC_NULL SQL_SRVC_DEFAULT SQL_SRVC_ROW_SUBQUERY

SQL_SQL92_STRING_FUNCTIONS(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the string scalar functions that are supported by the driver and the associated data source, as defined in SQL-92. The following bitmasks are used to determine which string functions are supported:

SQL_SSF_CONVERT SQL_SSF_LOWERS SQL_SSF_UPPERS SQL_SSF_SUBSTRING SQL_SSF_TRANSLATES SQL_SSF_TRIM_BOTH SQL_SSF_TRIM_LEADING SQL_SSF_TRIM_TRAILING

SQL_SQL92_VALUE_EXPRESSIONS(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the value expressions supported, as defined in SQL-92. The SQL-92 or FIPS conformance level at which this feature must be supported is shown in parentheses next to each bitmask.

The following bitmasks are used to determine which options are supported by the data source:

SQL_SVE_CASE (Intermediate level) SQL_SVE_CAST (FIPS Transitional level) SQL_SVE_COALESCE (Intermediate level) SQL_SVE_NULLIF (Intermediate level)

SQL_STANDARD_CLI_CONFORMANCE(ODBC 3.0)
An SQLUINTEGER bitmask enumerating the CLI standard or standards to which the driver conforms. The following bitmasks are used to determine which levels the driver complies with:

SQL_SCC_XOPEN_CLI_VERSION1: The driver complies with the Open Group CLI version 1.

SQL_SCC_ISO92_CLI: The driver complies with the ISO 92 CLI.

SQL_STATIC_CURSOR_ATTRIBUTES1(ODBC 3.0)
An SQLUINTEGER bitmask that describes the attributes of a static cursor that are supported by the driver. This bitmask contains the first subset of attributes; for the second subset, see SQL_STATIC_CURSOR_ATTRIBUTES2.

The following bitmasks are used to determine which attributes are supported:
SQL_CA1_NEXTSQL_CA1_ABSOLUTESQL_CA1_RELATIVESQL_CA1_BOOKMARKSQL_CA1_LOCK_
NO_CHANGEDSQL_CA1_LOCK_EXCLUSIVESQL_CA1_LOCK_UNLOCKSQL_CA1_POS_POSITIONSQL
_CA1_POS_UPDATESQL_CA1_POS_DELETESQL_CA1_POS_REFRESHSQL_CA1_POSITIONED_UP
DATESQL_CA1_POSITIONED_DELETESQL_CA1_SELECT_FOR_UPDATESQL_CA1_BULK_ADDSQL
_CA1_BULK_UPDATE_BY_BOOKMARKSQL_CA1_BULK_DELETE_BY_BOOKMARKSQL_CA1_BULK_F
ETCH_BY_BOOKMARK

For descriptions of these bitmasks, see SQL_DYNAMIC_CURSOR_ATTRIBUTES1 (and substitute "static cursor" for "dynamic cursor" in the descriptions).

An SQL-92 Intermediate level–conformant driver will usually return the SQL_CA1_NEXT, SQL_CA1bsolute, and SQL_CA1_RELATIVE options as supported, because the driver supports scrollable cursors through the embedded SQL FETCH statement. Because this does not directly determine the underlying SQL support, however, scrollable cursors may not be supported, even for an SQL-92 Intermediate level–conformant driver.

SQL_STATIC_CURSOR_ATTRIBUTES2(ODBC 3.0)

An SQLUINTEGER bitmask that describes the attributes of a static cursor that are supported by the driver. This bitmask contains the second subset of attributes; for the first subset, see SQL_STATIC_CURSOR_ATTRIBUTES1.

The following bitmasks are used to determine which attributes are supported:

SQL_CA2_READ_ONLY_CONCURRENCYSQL_CA2_LOCK_CONCURRENCYSQL_CA2_OPT_ROWVER
_CONCURRENCYSQL_CA2_OPT_VALUES_CONCURRENCYSQL_CA2_SENSITIVITY_ADDITIONSSQ
L_CA2_SENSITIVITY_DELETIONSSQL_CA2_SENSITIVITY_UPDATESQL_CA2_MAX_ROWS_SELE
CTSQL_CA2_MAX_ROWS_INSERTSQL_CA2_MAX_ROWS_DELETESQL_CA2_MAX_ROWS_UPDATE
SQL_CA2_MAX_ROWS_CATALOGSQL_CA2_MAX_ROWS_AFFECTS_ALLSQL_CA2_CRC_EXACTSQL
_CA2_CRC_APPROXIMATESQL_CA2_SIMULATE_NON_UNIQUESQL_CA2_SIMULATE_TRY_UNIQUE
SQL_CA2_SIMULATE_UNIQUE

For descriptions of these bitmasks, see SQL_DYNAMIC_CURSOR_ATTRIBUTES2 (and substitute "static cursor" for "dynamic cursor" in the descriptions).

SQL_STRING_FUNCTIONS(ODBC 1.0)

Note: The information type was introduced in ODBC 1.0; each bitmask is labeled with the version in which it was introduced.

An SQLUINTEGER bitmask enumerating the scalar string functions supported by the driver and associated data source.

The following bitmasks are used to determine which string functions are supported:

SQL_FN_STR_ASCII (ODBC 1.0)SQL_FN_STR_BIT_LENGTH (ODBC 3.0)SQL_FN_STR_CHAR
(ODBC 1.0)SQL_FN_STR_CHAR_LENGTH (ODBC 3.0)SQL_FN_STR_CHARACTER_LENGTH (ODBC
3.0)SQL_FN_STR_CONCAT (ODBC 1.0)SQL_FN_STR_DIFFERENCE (ODBC
2.0)SQL_FN_STR_INSERT (ODBC 1.0)SQL_FN_STR_LCASE (ODBC 1.0)SQL_FN_STR_LEFT
(ODBC 1.0)SQL_FN_STR_LENGTH (ODBC 1.0)SQL_FN_STR_LOCATE (ODBC 1.0)
SQL_FN_STR_LTRIM (ODBC 1.0)SQL_FN_STR_OCTET_LENGTH (ODBC 3.0)
SQL_FN_STR_POSITION (ODBC 3.0)SQL_FN_STR_REPEAT (ODBC 1.0)SQL_FN_STR_REPLACE
(ODBC 1.0)SQL_FN_STR_RIGHT (ODBC 1.0)SQL_FN_STR_RTRIM (ODBC
1.0)SQL_FN_STR_SOUNDEX (ODBC 2.0)SQL_FN_STR_SPACE (ODBC
2.0)SQL_FN_STR_SUBSTRING (ODBC 1.0)SQL_FN_STR_UCASE (ODBC 1.0)

If an application can call the LOCATE scalar function with the string_exp1, string_exp2, and start arguments, the driver returns the SQL_FN_STR_LOCATE bitmask. If an application can call the LOCATE scalar function with only the string_exp1 and string_exp2 arguments, the driver returns the SQL_FN_STR_LOCATE_2 bitmask. Drivers that fully support the LOCATE scalar
function return both bitmasks.

(For more information, see String Functions in Appendix E, "Scalar Functions.")

**SQL_SUBQUERIES (ODBC 2.0)**
An SQLUINTEGER bitmask enumerating the predicates that support subqueries:

- SQL_SQ_CORRELATED_SUBQUERIES
- SQL_SQ_COMPARE
- SQL_SQ_EXIST
- SQL_SQ_IN
- SQL_SQ_QUANTIFIED

The SQL_SQ_CORRELATED_SUBQUERIES bitmask indicates that all predicates that support subqueries support correlated subqueries.

An SQL-92 Entry level–conformant driver will always return a bitmask in which all of these bits are set.

**SQL_SYSTEM_FUNCTIONS (ODBC 1.0)**
An SQLUINTEGER bitmask enumerating the scalar system functions supported by the driver and associated data source.

The following bitmasks are used to determine which system functions are supported:

- SQL_FN_SYS_DBNAME
- SQL_FN_SYS_IFNULL
- SQL_FN_SYS_USERNAME

**SQL_TABLE_TERM (ODBC 1.0)**
A character string with the data source vendor’s name for a table; for example, "table" or "file". This character string can be in upper, lower, or mixed case.

An SQL-92 Entry level–conformant driver will always return "table".

**SQL_TIMEDATE_ADD_INTERVALS (ODBC 2.0)**
An SQLUINTEGER bitmask enumerating the timestamp intervals supported by the driver and associated data source for the TIMESTAMPADD scalar function.

The following bitmasks are used to determine which intervals are supported:

- SQL_FN_TSI_FRAC_SECOND
- SQL_FN_TSI_SECOND
- SQL_FN_TSI_MINUTE
- SQL_FN_TSI_HOUR
- SQL_FN_TSI_DAY
- SQL_FN_TSI_WEEK
- SQL_FN_TSI_MONTH
- SQL_FN_TSI_QUOTES
- SQL_FN_TSI_YEAR

An FIPS Transitional level–conformant driver will always return a bitmask in which all of these bits are set.

**SQL_TIMEDATE_DIFF_INTERVALS (ODBC 2.0)**
An SQLUINTEGER bitmask enumerating the timestamp intervals supported by the driver and associated data source for the TIMESTAMPDIFF scalar function.

The following bitmasks are used to determine which intervals are supported:

- SQL_FN_TSI_FRAC_SECOND
- SQL_FN_TSI_SECOND
- SQL_FN_TSI_MINUTE
- SQL_FN_TSI_HOUR
- SQL_FN_TSI_DAY
- SQL_FN_TSI_WEEK
- SQL_FN_TSI_MONTH
- SQL_FN_TSI_QUOTES
- SQL_FN_TSI_YEAR

An FIPS Transitional level–conformant driver will always return a bitmask in which all of these bits are set.

**SQL_TIMEDATE_FUNCTIONS (ODBC 1.0)**
Note: The information type was introduced in ODBC 1.0; each bitmask is labeled with the version in which it was introduced.
An SQLUINTEGER bitmask enumerating the scalar date and time functions supported by the driver and associated data source.

The following bitmasks are used to determine which date and time functions are supported:

- `SQL_FN_TD_CURRENT_DATE` (ODBC 3.0)
- `SQL_FN_TD_CURRENT_TIME` (ODBC 3.0)
- `SQL_FN_TD_CURRENT_TIMESTAMP` (ODBC 3.0)
- `SQL_FN_TD_CURDATE` (ODBC 1.0)
- `SQL_FN_TD_CURTIME` (ODBC 1.0)
- `SQL_FN_TD_DAYNAME` (ODBC 2.0)
- `SQL_FN_TD_DAYOFMONTH` (ODBC 1.0)
- `SQL_FN_TD_DAYOFYEAR` (ODBC 1.0)
- `SQL_FN_TD_EXTRACT` (ODBC 3.0)
- `SQL_FN_TD_HOUR` (ODBC 1.0)
- `SQL_FN_TD_MINUTE` (ODBC 1.0)
- `SQL_FN_TD_MONTH` (ODBC 1.0)
- `SQL_FN_TD_MONTHNAME` (ODBC 2.0)
- `SQL_FN_TD_NOW` (ODBC 1.0)
- `SQL_FN_TD_QUARTER` (ODBC 1.0)
- `SQL_FN_TD_SECOND` (ODBC 1.0)
- `SQL_FN_TD_TIMESTAMPADD` (ODBC 2.0)
- `SQL_FN_TD_TIMESTAMDIFF` (ODBC 2.0)
- `SQL_FN_TD_WEEK` (ODBC 1.0)
- `SQL_FN_TD_YEAR` (ODBC 1.0)

An SQLUSMALLINT value describing the transaction support in the driver or data source:

- `SQL_TC_NONE` = Transactions not supported. (ODBC 1.0)
- `SQL_TC_DML` = Transactions can contain only Data Manipulation Language (DML) statements (`SELECT`, INSERT, UPDATE, DELETE). Data Definition Language (DDL) statements encountered in a transaction cause an error. (ODBC 1.0)
- `SQL_TC_DDL_COMMIT` = Transactions can contain only DML statements. DDL statements (`CREATE TABLE`, DROP INDEX, and so on) encountered in a transaction cause the transaction to be committed. (ODBC 2.0)
- `SQL_TC_DDL_IGNORE` = Transactions can contain only DML statements. DDL statements encountered in a transaction are ignored. (ODBC 2.0)
- `SQL_TC_ALL` = Transactions can contain DDL statements and DML statements in any order. (ODBC 1.0)

(Again support of transactions is mandatory in SQL-92, an SQL-92 conformant driver [any level] will never return `SQL_TC_NONE`.)

An SQLUINTEGER bitmask enumerating the transaction isolation levels available from the driver or data source.

The following bitmasks are used together with the flag to determine which options are supported:

- `SQL_TXN_READ_UNCOMMITTED`
- `SQL_TXN_READ_COMMITTED`
- `SQL_TXN_REPEATABLE_READ`
- `SQL_TXN_SERIALIZABLE`

For descriptions of these isolation levels, see the description of SQL_DEFAULT_TXN_ISOLATION.

To set the transaction isolation level, an application calls `SQLSetConnectAttr` to set the SQL_ATTR_TXN_ISOLATION attribute. For more information, see `SQLSetConnectAttr Function`.

An SQL-92 Entry level–conformant driver will always return `SQL_TXN_SERIALIZABLE` as supported. A FIPS Transitional level–conformant driver will always return all of these options as...
**SQL_UNION (ODBC 2.0)**

An SQLUINTEGER bitmask enumerating the support for the `UNION` clause:

- **SQL_U_UNION** = The data source supports the `UNION` clause.
- **SQL_U_UNION_ALL** = The data source supports the `ALL` keyword in the `UNION` clause. *(SQLGetInfo returns both SQL_U_UNION and SQL_U_UNION_ALL in this case.)*

An SQL-92 Entry level–conformant driver will always return both of these options as supported.

**SQL_USER_NAME (ODBC 1.0)**

A character string with the name used in a particular database, which can be different from the login name.

**SQL_XOPEN_CLI_YEAR (ODBC 3.0)**

A character string that indicates the year of publication of the Open Group specification with which the version of the ODBC Driver Manager fully complies.

**SQL_ACCESSIBLE_PROCEDURES (ODBC 1.0)**

A character string: "Y" if the user can execute all procedures returned by `SQLProcedures`; "N" if there may be procedures returned that the user cannot execute.

**SQL_ACCESSIBLE_TABLES (ODBC 1.0)**

A character string: "Y" if the user is guaranteed `SELECT` privileges to all tables returned by `SQLTables`; "N" if there may be tables returned that the user cannot access.

**SQL_ACTIVE_ENVIRONMENTS (ODBC 3.0)**

An SQLUSMALLINT value that specifies the maximum number of active environments that the driver can support. If there is no specified limit or the limit is unknown, this value is set to zero.

**SQL_AGGREGATE_FUNCTIONS (ODBC 3.0)**

An SQLUINTEGER bitmask enumerating support for aggregation functions:

- SQL_AF_ALL
- SQL_AF_AVG
- SQL_AF_COUNT
- SQL_AF_DISTINCT
- SQL_AF_MAX
- SQL_AF_MIN
- SQL_AF_SUM

An SQL-92 Entry level–conformant driver will always return all of these options as supported.

**SQL_ALTER_DOMAIN (ODBC 3.0)**

An SQLUINTEGER bitmask enumerating the clauses in the `ALTER DOMAIN` statement, as defined in SQL-92, supported by the data source. An SQL-92 Full level–compliant driver will always return all of the bitmasks. A return value of "0" means that the `ALTER DOMAIN` statement is not supported.

The SQL-92 or FIPS conformance level at which this feature must be supported is shown in parentheses next to each bitmask.

The following bitmasks are used to determine which clauses are supported:

- **SQL_AD_ADD_DOMAIN_CONSTRAINT** = Adding a domain constraint is supported (Full level)
- **SQL_AD_ADD_DOMAIN_DEFAULT** = `<alter domain> <set domain default clause>` is supported (Full level)
- **SQL_AD_CONSTRAINT_NAME_DEFINITION** = `<constraint name definition clause>` is supported for naming domain constraint (Intermediate level)
- **SQL_AD_DROP_DOMAIN_CONSTRAINT** = `<drop domain constraint clause>` is supported (Full level)
SQL_AD_DROP_DOMAIN_DEFAULT = <alter domain> <drop domain default clause> is supported (Full level)

The following bits specify the supported <constraint attributes> if <add domain constraint> is supported (the SQL_AD_ADD_DOMAIN_CONSTRAINT bit is set):

SQL_AD_ADD_CONSTRAINT_DEFERRABLE (Full level)
SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE (Full level)
SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED (Full level)
SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE (Full level)

SQL_ALTER_TABLE(ODBC 2.0)
An SQLUINTEGER bitmask enumerating the clauses in the ALTER TABLE statement supported by the data source.

The SQL-92 or FIPS conformance level at which this feature must be supported is shown in parentheses next to each bitmask.

The following bitmasks are used to determine which clauses are supported:

SQL_AT_ADD_COLUMN_COLLATION = <add column> clause is supported, with facility to specify column collation (Full level) (ODBC 3.0)
SQL_AT_ADD_COLUMN_DEFAULT = <add column> clause is supported, with facility to specify column defaults (FIPS Transitional level) (ODBC 3.0)
SQL_AT_ADD_COLUMN_SINGLE = <add column> is supported (FIPS Transitional level) (ODBC 3.0)
SQL_AT_ADD_CONSTRAINT = <add column> clause is supported, with facility to specify column constraints (FIPS Transitional level) (ODBC 3.0)
SQL_AT_ADD_TABLE_CONSTRAINT = <add table constraint> clause is supported (FIPS Transitional level) (ODBC 3.0)
SQL_AT_CONSTRAINT_NAME_DEFINITION = <constraint name definition> is supported for naming column and table constraints (Intermediate level) (ODBC 3.0)
SQL_AT_DROP_COLUMN_CASCADE = <drop column> CASCADE is supported (FIPS Transitional level) (ODBC 3.0)
SQL_AT_DROP_COLUMN_DEFAULT = <alter column> <drop column default clause> is supported (Intermediate level) (ODBC 3.0)
SQL_AT_DROP_COLUMN_RESTRICT = <drop column> RESTRICT is supported (FIPS Transitional level) (ODBC 3.0)
SQL_AT_SET_COLUMN_DEFAULT = <alter column> <set column default clause> is supported (Intermediate level) (ODBC 3.0)

The following bits specify the support <constraint attributes> if specifying column or table constraints is supported (the SQL_AT_ADD_CONSTRAINT bit is set):
SQL_AT_CONSTRAINT_INITIALLY_DEFERRED (Full level) (ODBC 3.0)
SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE (Full level) (ODBC 3.0)
SQL_AT_CONSTRAINT_DEFERRABLE (Full level) (ODBC 3.0)
SQL_AT_CONSTRAINT_NON_DEFERRABLE (Full level) (ODBC 3.0)

SQL_ASYNC_MODE (ODBC 3.0)
An SQUINTEGER value that indicates the level of asynchronous support in the driver:

SQL_AM_CONNECTION = Connection level asynchronous execution is supported. Either all statement handles associated with a given connection handle are in asynchronous mode or all are in synchronous mode. A statement handle on a connection cannot be in asynchronous mode while another statement handle on the same connection is in synchronous mode, and vice versa.

SQL_AM_STATEMENT = Statement level asynchronous execution is supported. Some statement handles associated with a connection handle can be in asynchronous mode, whereas other statement handles on the same connection are in synchronous mode.

SQL_AM_NONE = Asynchronous mode is not supported.

SQL_BATCH_ROW_COUNT (ODBC 3.0)
An SQUINTEGER bitmask enumerating the behavior of the driver with respect to the availability of row counts. The following bitmasks are used together with the information type:

SQL_BRC_ROLLED_UP = Row counts for consecutive INSERT, DELETE, or UPDATE statements are rolled up into one. If this bit is not set, row counts are available for each statement.

SQL_BRC_PROCEDURES = Row counts, if any, are available when a batch is executed in a stored procedure. If row counts are available, they can be rolled up or individually available, depending on the SQL_BRC_ROLLED_UP bit.

SQL_BRC_EXPLICIT = Row counts, if any, are available when a batch is executed directly by calling SQLExecute or SQLExecDirect. If row counts are available, they can be rolled up or individually available, depending on the SQL_BRC_ROLLED_UP bit.

SQLGetInfo returns lists of supported options as an SQUINTEGER bitmask in *InfoValuePtr. The bitmask for each option is used together with the flag to determine whether the option is supported.

For example, an application could use the following code to determine whether the SUBSTRING scalar function is supported by the driver associated with the connection.

For another example of using SQLGetInfo, see SQLTables Function.

```c
SQUINTEGER fFuncs;

SQLGetInfo(hdbc,
    SQL_STRING_FUNCTIONS,
    (SQLPOINTER)&fFuncs,
    sizeof(fFuncs),
    NULL);

// SUBSTRING supported
if (fFuncs & SQL_FN_STR_SUBSTRING)
    ; // do something
```
Related Functions

Returning the setting of a connection attribute  
SQLGetConnectAttr Function

Determining whether a driver supports a function  
SQLGetFunctions Function

Returning the setting of a statement attribute  
SQLGetStmtAttr Function

Returning information about a data source’s data types  
SQLGetTypeInfo Function

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLGetStmtAttr Function

Conformance

Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary

SQLGetStmtAttr returns the current setting of a statement attribute.

Note

For more information about what the Driver Manager maps this function to when an ODBC 3.x application is working with an ODBC 2.x driver, see Mapping Replacement Functions for Backward Compatibility of Applications.

Syntax

```c
SQLRETURN SQLGetStmtAttr(
    SQLHSTMT StatementHandle,
```
Arguments

StatementHandle
    [Input] Statement handle.

Attribute
    [Input] Attribute to retrieve.

ValuePtr
    [Output] Pointer to a buffer in which to return the value of the attribute specified in Attribute. If ValuePtr is NULL, StringLengthPtr will still return the total number of bytes (excluding the null-termination character for character data) available to return in the buffer pointed to by ValuePtr.

BufferLength
    [Input] If Attribute is an ODBC-defined attribute and ValuePtr points to a character string or a binary buffer, this argument should be the length of *ValuePtr. If Attribute is an ODBC-defined attribute and *ValuePtr is an integer, BufferLength is ignored. If the value returned in *ValuePtr is a Unicode string (when calling SQLGetStmtAttrW), the BufferLength argument must be an even number.

If Attribute is a driver-defined attribute, the application indicates the nature of the attribute to the Driver Manager by setting the BufferLength argument. BufferLength can have the following values:

- If *ValuePtr is a pointer to a character string, then BufferLength is the length of the string or SQL_NTS.

- If *ValuePtr is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in BufferLength. This places a negative value in BufferLength.

- If *ValuePtr is a pointer to a value other than a character string or binary string, then BufferLength should have the value SQL_IS_POINTER.

- If *ValuePtr is contains a fixed-length data type, then BufferLength is either SQL_IS_INTEGER or SQL_IS_UINTINTEGER, as appropriate.

StringLengthPtr
    [Output] A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character) available to return in *ValuePtr. If ValuePtr is a null pointer, no length is returned. If the attribute value is a character string, and the number of bytes available to return is greater than or equal to BufferLength, the data in *ValuePtr is truncated to BufferLength minus the length of a null-termination character and is null-terminated by the driver.
Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When `SQLGetStmtAttr` returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling `SQLGetDiagRec` with a `HandleType` of SQL_HANDLE_STMT and a `Handle` of `StatementHandle`. The following table lists the SQLSTATE values commonly returned by `SQLGetStmtAttr` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right</td>
<td>The data returned in <code>*ValuePtr</code> was truncated to be <code>BufferLength</code> minus the length of a null-termination character. The length of the untruncated string value is returned in <code>*StringLengthPtr</code>. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>The argument <code>Attribute</code> was SQL_ATTR_ROW_NUMBER and the cursor was not open, or the cursor was positioned before the start of the result set or after the end of the result set.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the argument <code>MessageText</code> describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when the <code>SQLGetStmtAttr</code> function was called.</td>
</tr>
</tbody>
</table>

(DM) An asynchronously executing function was called for the `StatementHandle` and was still executing when this function was called.

(DM) `SQLExecute`, `SQLExecDirect`, `SQLBulkOperations`, or `SQLSetPos` was called for the `StatementHandle` and returned SQL_NEED_DATA. This
function was called before data was sent for all data-at-execution parameters or columns.

HY013 Memory management error
The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.

HY090 Invalid string or buffer length
*(DM)* *ValuePtr* is a character string, and BufferLength was less than zero, but not equal to SQL_NTS.

HY092 Invalid attribute/options identifier
The value specified for the argument Attribute was not valid for the version of ODBC supported by the driver.

HY109 Invalid cursor position
The Attribute argument was SQL_ATTR_ROW_NUMBER and the row had been deleted or could not be fetched.

HY117 Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.
*(DM)* For more information about suspended state, see SQLEndTran Function.

HYC00 Optional feature not implemented
The value specified for the argument Attribute was a valid ODBC statement attribute for the version of ODBC supported by the driver, but was not supported by the driver.

HYT01 Connection timeout expired
The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.

IM001 Driver does not support this function
*(DM)* The driver corresponding to the StatementHandle does not support the function.

Comments

For general information about statement attributes, see Statement Attributes.

A call to SQLGetStmtAttr returns in *ValuePtr* the value of the statement attribute specified in Attribute. That value can either be a SQLULEN value or a null-terminated character string. If the value is a SQLULEN value, some drivers may only write the lower 32-bit or 16-bit of a buffer and leave the higher-order bit unchanged. Therefore, applications should use a buffer of SQLULEN and initialize the value to 0 before calling this function. Also, the BufferLength and StringLengthPtr arguments are not used. If the value is a null-terminated string, the application specifies the maximum length of that string in the BufferLength argument, and the driver returns the length of that string in the *StringLengthPtr* buffer.

To allow applications calling SQLGetStmtAttr to work with ODBC 2.x drivers, a call to SQLGetStmtAttr is mapped in the Driver Manager to SQLGetStmtOption.

The following statement attributes are read-only, so can be retrieved by SQLGetStmtAttr, but not set
by SQLSetStmtAttr:

- SQL_ATTR_IMP_PARAM_DESC
- SQL_ATTR_IMP_ROW_DESC
- SQL_ATTR_ROW_NUMBER

For a list of attributes that can be set and retrieved, see SQLSetStmtAttr.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returning the setting of a connection attribute</td>
<td>SQLGetConnectAttr Function</td>
</tr>
<tr>
<td>Setting a connection attribute</td>
<td>SQLSetConnectAttr Function</td>
</tr>
<tr>
<td>Setting a statement attribute</td>
<td>SQLSetStmtAttr Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLGetStmtOption Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

Summary
In ODBC 3.x, the ODBC 2.0 function SQLGetStmtOption has been replaced by SQLSetStmtAttr. For more information, see SQLSetStmtAttr.

Note
For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.
SQLGetTypeInfo Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLGetTypeInfo returns information about data types supported by the data source. The driver returns the information in the form of an SQL result set. The data types are intended for use in Data Definition Language (DDL) statements.

Important
Applications must use the type names returned in the TYPE_NAME column of the SQLGetTypeInfo result set in ALTER TABLE and CREATE TABLE statements. SQLGetTypeInfo may return more than one row with the same value in the DATA_TYPE column.

Syntax

```
SQLRETURN SQLGetTypeInfo(
    SQLHSTMT StatementHandle,
    SQLSMALLINT DataType);
```

Arguments

StatementHandle
[Input] Statement handle for the result set.

DataType
[Input] The SQL data type. This must be one of the values in the SQL Data Types section of Appendix D: Data Types, or a driver-specific SQL data type. SQL_ALL_TYPES specifies that information about all data types should be returned.

Returns
When SQLGetTypeInfo returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLGetTypeInfo and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
</tr>
<tr>
<td></td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed</td>
</tr>
<tr>
<td></td>
<td>A specified statement attribute was invalid because of implementation working conditions, so a similar value was temporarily substituted. (Call SQLGetStmtAttr to determine the temporarily substituted value.) The substitute value is valid for the StatementHandle until the cursor is closed. The statement attributes that can be changed are: SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, SQL_ATTR_KEYSET_SIZE, SQL_ATTR_MAX_LENGTH, SQL_ATTR_MAX_ROWS, SQL_ATTR_QUERY_TIMEOUT, and SQL_ATTR_SIMULATE_CURSOR. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
</tr>
<tr>
<td></td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
</tr>
<tr>
<td></td>
<td>A cursor was open on the StatementHandle, and SQLFetch or SQLFetchScroll had been called. This error is returned by the Driver Manager if SQLFetch or SQLFetchScroll has not returned SQL_NO_DATA, and is returned by the driver if SQLFetch or SQLFetchScroll has returned SQL_NO_DATA. A result set was open on the StatementHandle, but SQLFetch or SQLFetchScroll had not been called.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
</tr>
<tr>
<td></td>
<td>The transaction was rolled back due to a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
</tr>
<tr>
<td></td>
<td>The associated connection failed during the execution of this function and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
</tr>
</tbody>
</table>
|          | An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was
The error message returned by `SQLGetDiagRec` in the `*MessageText` buffer describes the error and its cause.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY004</td>
<td>Invalid SQL data type</td>
<td>The value specified for the argument <code>DataType</code> was neither a valid ODBC SQL data type identifier nor a driver-specific data type identifier supported by the driver.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <code>StatementHandle</code>, then the function was called and, before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code>. Then the function was called again on the <code>StatementHandle</code>. The function was called and, before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when the <code>SQLGetTypeInfo</code> function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for the <code>StatementHandle</code> and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. (DM) An asynchronously executing function (not this one) was called for the <code>StatementHandle</code> and was still executing when this function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for the <code>StatementHandle</code> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see <code>SQLEndTran Function</code>.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The combination of the current settings of the <code>SQL_ATTR_CONCURRENCY</code> and <code>SQL_ATTR_CURSOR_TYPE</code></td>
</tr>
</tbody>
</table>
statement attributes was not supported by the driver or data source.

The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks.

HYT00  Timeout expired  The query timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.

HYT01  Connection timeout expired  The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.

IM001  Driver does not support this function  (DM) The driver corresponding to the StatementHandle does not support the function.

IM017  Polling is disabled in asynchronous notification mode  Whenever the notification model is used, polling is disabled.

IM018  SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.  If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.

Comments

**SQLGetTypeInfo** returns the results as a standard result set, ordered by DATA_TYPE and then by how closely the data type maps to the corresponding ODBC SQL data type. Data types defined by the data source take precedence over user-defined data types. Consequently, the sort order is not necessarily consistent but can be generalized as DATA_TYPE first, followed by TYPE_NAME, both ascending. For example, suppose that a data source defined INTEGER and COUNTER data types, where COUNTER is auto-incrementing, and that a user-defined data type WHOLENUM has also been defined. These would be returned in the order INTEGER, WHOLENUM, and COUNTER, because WHOLENUM maps closely to the ODBC SQL data type SQL_INTEGER, while the auto-incrementing data type, even though supported by the data source, does not map closely to an ODBC SQL data type. For information about how this information might be used, see DDL Statements.

If the *DataType* argument specifies a data type which is valid for the version of ODBC supported by the driver, but is not supported by the driver, then it will return an empty result set.

**Note**

For more information about the general use, arguments, and returned data of ODBC catalog functions, see Catalog Functions.
The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRECISION</td>
<td>COLUMN_SIZE</td>
</tr>
<tr>
<td>MONEY</td>
<td>FIXED_PREC_SCALE</td>
</tr>
<tr>
<td>AUTO_INCREMENT</td>
<td>AUTO_UNIQUE_VALUE</td>
</tr>
</tbody>
</table>

The following columns have been added to the results set returned by `SQLGetTypeInfo` for ODBC 3.x:

- SQL_DATA_TYPE
- INTERVAL_PRECISION
- SQL_DATETIME_SUB
- NUM_PREC_RADIX

The following table lists the columns in the result set. Additional columns beyond column 19 (INTERVAL_PRECISION) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set rather than specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.

**Note**

`SQLGetTypeInfo` might not return all data types. For example, a driver might not return user-defined data types. Applications can use any valid data type, regardless of whether it is returned by `SQLGetTypeInfo`. The data types returned by `SQLGetTypeInfo` are those supported by the data source. They are intended for use in Data Definition Language (DDL) statements. Drivers can return result-set data using data types other than the types returned by `SQLGetTypeInfo`. In creating the result set for a catalog function, the driver might use a data type that is not supported by the data source.

<table>
<thead>
<tr>
<th>Column name (ODBC 2.0)</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_NAME</td>
<td>1</td>
<td>Varchar</td>
<td>Data source–dependent data-type name; for example, &quot;CHAR()&quot;, &quot;VARCHAR()&quot;, &quot;MONEY&quot;, &quot;LONG VARBINARY&quot;, or &quot;CHAR ( ) FOR BIT DATA&quot;. Applications must use this name in <code>CREATE TABLE</code> and <code>ALTER TABLE</code> statements.</td>
</tr>
<tr>
<td>DATA_TYPE (ODBC 2.0)</td>
<td>2</td>
<td>Smallint</td>
<td>SQL data type. This can be an ODBC SQL data type or a driver-specific SQL data type. For datetime or interval data types, this column returns the concise...</td>
</tr>
</tbody>
</table>
data type (such as SQL_TYPE_TIME or SQL_INTERVAL_YEAR_TO_MONTH). For a list of valid ODBC SQL data types, see SQL Data Types in Appendix D: Data Types. For information about driver-specific SQL data types, see the driver’s documentation.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMN_SIZE (ODBC 2.0)</td>
<td>Integer</td>
<td>The maximum column size that the server supports for this data type. For numeric data, this is the maximum precision. For string data, this is the length in characters. For datetime data types, this is the length in characters of the string representation (assuming the maximum allowed precision of the fractional seconds component). NULL is returned for data types where column size is not applicable. For interval data types, this is the number of characters in the character representation of the interval literal (as defined by the interval leading precision; see Interval Data Type Length in Appendix D: Data Types). For more information on column size, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.</td>
</tr>
<tr>
<td>LITERAL_PREFIX (ODBC 2.0)</td>
<td>Varchar</td>
<td>Character or characters used to prefix a literal; for example, a single quotation mark (’) for character data types or 0x for binary data types; NULL is returned for data types where a literal prefix is not applicable.</td>
</tr>
<tr>
<td>LITERAL_SUFFIX (ODBC 2.0)</td>
<td>Varchar</td>
<td>Character or characters used to terminate a literal; for example, a single quotation mark (’) for character data types; NULL is returned for data types where a literal suffix is not applicable.</td>
</tr>
<tr>
<td>CREATE_PARAMS (ODBC 2.0)</td>
<td>Varchar</td>
<td>A list of keywords, separated by commas, corresponding to each parameter that the application may specify in parentheses when using the name that is returned in the TYPE_NAME field. The keywords in the list can be any of the following: length, precision, or scale. They appear in the order that the syntax requires them to be used. For example, CREATE_PARAMS for DECIMAL would be “precision, scale”; CREATE_PARAMS for VARCHAR would equal “length.” NULL is returned if there are no parameters for the data type definition; for example, INTEGER. The driver supplies the CREATE_PARAMS text in the language of the country/region where it is used.</td>
</tr>
<tr>
<td>NULLABLE (ODBC 2.0)</td>
<td>Smallint</td>
<td>Whether the data type accepts a NULL value: SQL_NO_NULLS if the data type does not accept NULL.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Precision</td>
<td>Data Type</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------</td>
<td>------------------</td>
</tr>
<tr>
<td>NULL values.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CASE_SENSITIVE (ODBC 2.0)</td>
<td>8</td>
<td>Smallint not NULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEARCHABLE (ODBC 2.0)</td>
<td>9</td>
<td>Smallint not NULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_ATTRIBUTE (ODBC 2.0)</td>
<td>10</td>
<td>Smallint</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIXED_PREC_SCALE (ODBC 2.0)</td>
<td>11</td>
<td>Smallint not NULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AUTO_UNIQUE_VALUE (ODBC 2.0)</td>
<td>Smallint</td>
<td>Whether the data type is autoincrementing: SQL_TRUE if the data type is autoincrementing. SQL_FALSE if the data type is not autoincrementing. NULL is returned if the attribute is not applicable to the data type or the data type is not numeric. An application can insert values into a column having this attribute, but typically cannot update the values in the column. When an insert is made into an auto-increment column, a unique value is inserted into the column at insert time. The increment is not defined, but is data source-specific. An application should not assume that an auto-increment column starts at any particular point or increments by any particular value.</td>
</tr>
<tr>
<td>LOCAL_TYPE_NAME (ODBC 2.0)</td>
<td>Varchar</td>
<td>Localized version of the data source–dependent name of the data type. NULL is returned if a localized name is not supported by the data source. This name is intended for display only, such as in dialog boxes.</td>
</tr>
<tr>
<td>MINIMUM_SCALE (ODBC 2.0)</td>
<td>Smallint</td>
<td>The minimum scale of the data type on the data source. If a data type has a fixed scale, the MINIMUM_SCALE and MAXIMUM_SCALE columns both contain this value. For example, an SQL_TYPE_TIMESTAMP column might have a fixed scale for fractional seconds. NULL is returned where scale is not applicable. For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.</td>
</tr>
<tr>
<td>MAXIMUM_SCALE (ODBC 2.0)</td>
<td>Smallint</td>
<td>The maximum scale of the data type on the data source. NULL is returned where scale is not applicable. If the maximum scale is not defined separately on the data source, but is instead defined to be the same as the maximum precision, this column contains the same value as the COLUMN_SIZE column. For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.</td>
</tr>
<tr>
<td>SQL_DATA_TYPE (ODBC 3.0)</td>
<td>Smallint</td>
<td>The value of the SQL data type as it appears in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column, except for interval and datetime data types. For interval and datetime data types, the SQL_DATA_TYPE field in the result set will return SQL_INTERVAL or SQL_DATETIME, and the</td>
</tr>
</tbody>
</table>
The SQL_DATETIME_SUB field will return the subcode for the specific interval or datetime data type. (See Appendix D: Data Types.)

| SQL_DATETIME_SUB (ODBC 3.0) | 17 | Smallint | When the value of SQL_DATA_TYPE is SQL_DATETIME or SQL_INTERVAL, this column contains the datetime/interval subcode. For data types other than datetime and interval, this field is NULL. For interval or datetime data types, the SQL_DATA_TYPE field in the result set will return SQL_INTERVAL or SQL_DATETIME, and the SQL_DATETIME_SUB field will return the subcode for the specific interval or datetime data type. (See Appendix D: Data Types.) |

| NUM_PREC_RADIX (ODBC 3.0) | 18 | Integer | If the data type is an approximate numeric type, this column contains the value 2 to indicate that COLUMN_SIZE specifies a number of bits. For exact numeric types, this column contains the value 10 to indicate that COLUMN_SIZE specifies a number of decimal digits. Otherwise, this column is NULL. |

| INTERVAL_PRECISION (ODBC 3.0) | 19 | Smallint | If the data type is an interval data type, then this column contains the value of the interval leading precision. (See Interval Data Type Precision in Appendix D: Data Types.) Otherwise, this column is NULL. |

Attribute information can apply to data types or to specific columns in a result set. SQLGetTypeInfo returns information about attributes associated with data types; SQLColAttribute returns information about attributes associated with columns in a result set.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Returning information about a column in a result set</td>
<td>SQLColAttribute Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Returning information about a driver or data source</td>
<td>SQLGetInfo Function</td>
</tr>
</tbody>
</table>
SQLMoreResults Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ODBC

Summary
SQLMoreResults determines whether more results are available on a statement containing SELECT, UPDATE, INSERT, or DELETE statements and, if so, initializes processing for those results.

Syntax

```
SQLRETURN SQLMoreResults(
    SQLHSTMT StatementHandle);
```

Arguments

StatementHandle
[Input] Statement handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_NO_DATA, SQL_ERROR, SQL_INVALID_HANDLE, OR SQL_PARAM_DATA_AVAILABLE.

Diagnostics
When SQLMoreResults returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLMoreResults and explains each one in the context of this function; the notation "(DM)" precedes
the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value has changed</td>
<td>The value of a statement attribute changed as the batch was being processed. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back due to a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
</tbody>
</table>
| HY008 | Operation canceled | Asynchronous processing was enabled for the StatementHandle. The SQLMoreResults function was called and, before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle. Then the SQLMoreResults function was called again on the StatementHandle.

The SQLMoreResults function was called and, before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle from a different thread in a multithread application. |
| HY010 | Function sequence error | (DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLMoreResults function was called.

(DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called.

(DM) SQLExecute, SQLExecDirect, SQLBulkOperations,
or **SQLSetPos** was called for the *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see <a href="#">SQLEndTran Function</a>.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>, <code>SQL_ATTR_CONNECTION_TIMEOUT</code>.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the <em>StatementHandle</em> does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><strong>SQLCompleteAsync</strong> has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, <strong>SQLCompleteAsync</strong> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

### Comments

**SELECT** statements return result sets. **UPDATE**, **INSERT**, and **DELETE** statements return a count of affected rows. If any of these statements are batched, submitted with arrays of parameters (numbered in increasing parameter order, in the order that they appear in the batch), or in procedures, they can return multiple result sets or row counts. For information about batches of statements and arrays of parameters, see [Batches of SQL Statements](#) and [Arrays of Parameter Values](#).

After executing the batch, the application is positioned on the first result set. The application can call **SQLBindCol**, **SQLBulkOperations**, **SQLFetch**, **SQLGetData**, **SQLFetchScroll**, **SQLSetPos**, and all the metadata functions, on the first or any subsequent result sets, just as it would if there were just a single result set. Once it is done with the first result set, the application calls **SQLMoreResults** to move to the next result set. If another result set or count is available, **SQLMoreResults** returns SQL_SUCCESS and initializes the result set or count for additional processing. If any row count–generating statements appear in between result set–generating statements, they can be stepped over by calling **SQLMoreResults**. After calling **SQLMoreResults** for **UPDATE**, **INSERT**, or **DELETE** statements, an application can call **SQLRowCount**.
If there was a current result set with unfetched rows, `SQLMoreResults` discards that result set and makes the next result set or count available. If all results have been processed, `SQLMoreResults` returns SQL_NO_DATA. For some drivers, output parameters and return values are not available until all result sets and row counts have been processed. For such drivers, output parameters and return values become available when `SQLMoreResults` returns SQL_NO_DATA.

Any bindings that were established for the previous result set still remain valid. If the column structures are different for this result set, then calling `SQLFetch` or `SQLFetchScroll` may result in an error or truncation. To prevent this, the application has to call `SQLBindCol` with an `Option` of SQL_UNBIND to unbind all the column buffers. Alternatively, the application can call `SQLFreeStmt` with an `Option` of SQL_CLOSE to discard all the result sets and row counts that were available as a result of the execution of the batch. The statement handle returns to either the allocated or prepared state. Calling `SQLCancel` to cancel an asynchronously executing function when a batch has been executed and the statement handle is in the executed, cursor-positioned, or asynchronous state results in all the results sets and row counts generated by the batch being discarded if the cancel call was successful. The statement then returns to the prepared or allocated state.

If a batch of statements or a procedure mixes other SQL statements with `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements, these other statements do not affect `SQLMoreResults`.

For more information, see Multiple Results.

If a searched update, insert, or delete statement in a batch of statements does not affect any rows at the data source, `SQLMoreResults` returns SQL_SUCCESS. This is different from the case of a searched update, insert, or delete statement that is executed through `SQLExecDirect`, `SQLExecute`, or `SQLParamData`, which returns SQL_NO_DATA if it does not affect any rows at the data source. If an application calls `SQLRowCount` to retrieve the row count after a call to `SQLMoreResults` has not affected any rows, `SQLRowCount` will return SQL_NO_DATA.

For additional information about the valid sequencing of result-processing functions, see Appendix B: ODBC State Transition Tables.

For more information about SQL_PARAM_DATA_AVAILABLE and streamed output parameters, see Retrieving Output Parameters Using SQLGetData.

### Availability of Row Counts

When a batch contains multiple consecutive row count–generating statements, it is possible that these row counts are rolled up into just one row count. For example, if a batch has five insert statements, then certain data sources are capable of returning five individual row counts. Certain other data sources return only one row count that represents the sum of the five individual row counts.

When a batch contains a combination of result set–generating and row count–generating statements, row counts may or may not be available at all. The behavior of the driver with respect to the availability of row counts is enumerated in the SQL_BATCH_ROW_COUNT information type available through a call to `SQLGetInfo`. For example, suppose that the batch contains a `SELECT`, followed by two `INSERT`s and another `SELECT`. Then the following cases are possible:
The row counts corresponding to the two INSERT statements are not available at all. The first call to SQLMoreResults will position you on the result set of the second SELECT statement.

The row counts corresponding to the two INSERT statements are available individually. (A call to SQLGetInfo does not return the SQL_BRC_ROLLED_UP bit for the SQL_BATCH_ROW_COUNT information type.) The first call to SQLMoreResults will position you on the row count of the first INSERT, and the second call will position you on the row count of the second INSERT. The third call to SQLMoreResults will position you on the result set of the second SELECT statement.

The row counts corresponding to the two INSERTs are rolled up into one single row count that is available. (A call to SQLGetInfo returns the SQL_BRC_ROLLED_UP bit for the SQL_BATCH_ROW_COUNT information type.) The first call to SQLMoreResults will position you on the rolled-up row count, and the second call to SQLMoreResults will position you on the result set of the second SELECT.

Certain drivers make row counts available only for explicit batches and not for stored procedures.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Fetching part or all of a column of data</td>
<td>SQLGetData Function</td>
</tr>
</tbody>
</table>

**See Also**

ODBC API Reference
ODBC Header Files
Retrieving Output Parameters Using SQLGetData

© 2016 Microsoft

**SQLNativeSql Function**

**Conformance**

Version Introduced: ODBC 1.0 Standards Compliance: ODBC
Summary

SQLNativeSql returns the SQL string as modified by the driver. SQLNativeSql does not execute
the SQL statement.

Syntax

```c
SQLRETURN SQLNativeSql(
    SQLHDBC								ConnectionHandle,
    SQLCHAR	*						InStatementText,
    SQLINTEGER					TextLength1,
    SQLCHAR	*						OutStatementText,
    SQLINTEGER					BufferLength,
    SQLINTEGER	*			TextLength2Ptr);
```

Arguments

**ConnectionHandle**

[Input] Connection handle.

**InStatementText**

[Input] SQL text string to be translated.

**TextLength1**

[Input] Length in characters of *InStatementText text string.

**OutStatementText**

[Output] Pointer to a buffer in which to return the translated SQL string.

If *OutStatementText is NULL, *TextLength2Ptr will still return the total number of characters
(excluding the null-termination character for character data) available to return in the buffer
pointed to by *OutStatementText.

**BufferLength**

[Input] Number of characters in the *OutStatementText buffer. If the value returned in
*InStatementText is a Unicode string (when calling SQLNativeSqlW), the BufferLength
argument must be an even number.

**TextLength2Ptr**

[Output] Pointer to a buffer in which to return the total number of characters (excluding null-
termination) available to return in *OutStatementText. If the number of characters available to
return is greater than or equal to BufferLength, the translated SQL string in *OutStatementText
is truncated to BufferLength minus the length of a null-termination character.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.
Diagnostics

When SQLNativeSql returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_DBC and a Handle of ConnectionHandle. The following table lists the SQLSTATE values commonly returned by SQLNativeSql and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The buffer *OutStatementText was not large enough to return the entire SQL string, so the SQL string was truncated. The length of the untruncated SQL string is returned in *TextLength2Ptr. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
<td>The ConnectionHandle was not in a connected state.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format</td>
<td>*InStatementText contained an escape clause with an invalid date, time, or timestamp value.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>The cursor referred to in the statement was positioned before the start of the result set or after the end of the result set. This error may not be returned by a driver having a native DBMS cursor implementation.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>(DM) *InStatementText was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the ConnectionHandle and was still executing when this function was called.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
</tbody>
</table>
### Comments

The following are examples of what `SQLNativeSql` might return for the following input SQL string containing the scalar function `CONVERT`. Assume that the column `empid` is of type `INTEGER` in the data source:

```sql
SELECT { fn CONVERT (empid, SQL_SMALLINT) } FROM employee
```

A driver for Microsoft SQL Server might return the following translated SQL string:

```sql
SELECT convert (smallint, empid) FROM employee
```

A driver for ORACLE Server might return the following translated SQL string:

```sql
SELECT to_number (empid) FROM employee
```

A driver for Ingres might return the following translated SQL string:
Related Functions

None.

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLNumParams Function

Conformance

Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary

SQLNumParams returns the number of parameters in an SQL statement.

Syntax

```c
SQLRETURN SQLNumParams(
    SQLHSTMT StatementHandle,
    SQLSMALLINT * ParameterCountPtr);
```

Arguments

StatementHandle

[Input] Statement handle.

ParameterCountPtr

[Output] Pointer to a buffer in which to return the number of parameters in the statement.
Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
SQL_INVALID_HANDLE.

Diagnostics

When SQLNumParams returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLNumParams and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the StatementHandle. The SQLNumParams function was called and, before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle; the SQLNumParams function was then called again on the StatementHandle. Or, the SQLNumParams function was called and, before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) The function was called prior to calling SQLPrepare or SQLExecDirect for the StatementHandle. (DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLNumParams function was called.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information on suspended state, see <code>SQLEndTran Function</code>.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>, <code>SQL_ATTR_CONNECTION_TIMEOUT</code>.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the <code>StatementHandle</code> does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><code>SQLCompleteAsync</code> has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns <code>SQL_STILL_EXECUTING</code> and if notification mode is enabled, <code>SQLCompleteAsync</code> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

**Comments**

`SQLNumParams` can be called only after `SQLPrepare` has been called.

If the statement associated with `StatementHandle` does not contain parameters, `SQLNumParams` sets `*ParameterCountPtr` to 0.

The number of parameters returned by `SQLNumParams` is the same value as the `SQL_DESC_COUNT` field of the IPD.

For more information, see [Describing Parameters](#).
SQLNumResultCols Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLNumResultCols returns the number of columns in a result set.

Syntax

```
SQLRETURN SQLNumResultCols(
    SQLHSTMT StatementHandle,
    SQLSMALLINT * ColumnCountPtr);
```

Arguments

*StatementHandle*
[Input] Statement handle.

*ColumnCountPtr*
[Output] Pointer to a buffer in which to return the number of columns in the result set. This count does not include a bound bookmark column.
Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When `SQLNumResultCols` returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling `SQLGetDiagRec` with a `HandleType` of SQL_HANDLE_STMT and a `Handle` of StatementHandle. The following table lists the SQLSTATE values commonly returned by `SQLNumResultCols` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the StatementHandle. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the StatementHandle; the function was then called again on the StatementHandle. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the StatementHandle from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the <code>SQLNumResultCols</code> function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for the StatementHandle and returned</td>
</tr>
</tbody>
</table>
SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.

(DM) The function was called prior to calling `SQLPrepare` or `SQLExecDirect` for the `StatementHandle`.

(DM) An asynchronously executing function (not this one) was called for the `StatementHandle` and was still executing when this function was called.

(DM) `SQLExecute`, `SQLExecDirect`, `SQLBulkOperations`, or `SQLSetPos` was called for the `StatementHandle` and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

See `SQLPrepare Function` for details on when a statement handle can be freed.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
</tr>
<tr>
<td></td>
<td>The function call could not be processed because the</td>
</tr>
<tr>
<td></td>
<td>underlying memory objects could not be accessed,</td>
</tr>
<tr>
<td></td>
<td>possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction</td>
</tr>
<tr>
<td></td>
<td>state. Only disconnect and read-only functions are</td>
</tr>
<tr>
<td></td>
<td>allowed.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
</tr>
<tr>
<td></td>
<td>The connection timeout period expired before the data</td>
</tr>
<tr>
<td></td>
<td>source responded to the request. The connection timeout</td>
</tr>
<tr>
<td></td>
<td>period is set through <code>SQLSetConnectAttr</code>, `SQL_ATTR_</td>
</tr>
<tr>
<td></td>
<td>CONNECTION_TIMEOUT`.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
</tr>
<tr>
<td></td>
<td>(DM) The driver associated with the <code>StatementHandle</code></td>
</tr>
<tr>
<td></td>
<td>does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
</tr>
<tr>
<td></td>
<td>Whenever the notification model is used, polling is</td>
</tr>
<tr>
<td></td>
<td>disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><strong>SQLCompleteAsync</strong></td>
</tr>
<tr>
<td></td>
<td>has not been called to complete the previous</td>
</tr>
<tr>
<td></td>
<td>asynchronous operation on this handle.</td>
</tr>
<tr>
<td></td>
<td>If the previous function call on the handle returns</td>
</tr>
<tr>
<td></td>
<td>SQL_STILL_EXECUTING and if notification mode is</td>
</tr>
<tr>
<td></td>
<td>enabled, <strong>SQLCompleteAsync</strong> must be called on the</td>
</tr>
<tr>
<td></td>
<td>handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

`SQLNumResultCols` can return any SQLSTATE that can be returned by `SQLPrepare` or `SQLExecute` when called after `SQLPrepare` and before `SQLExecute`, depending on when the data source evaluates the SQL statement associated with the statement.
Comments

SQLNumResultCols can be called successfully only when the statement is in the prepared, executed, or positioned state.

If the statement associated with StatementHandle does not return columns, SQLNumResultCols sets *ColumnCountPtr to 0.

The number of columns returned by SQLNumResultCols is the same value as the SQL_DESC_COUNT field of the IRD.

For more information, see Was a Result Set Created? and How is Metadata Used?.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Returning information about a column in a result set</td>
<td>SQLColAttribute Function</td>
</tr>
<tr>
<td>Returning information about a column in a result set</td>
<td>SQLDescribeCol Function</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Fetching part or all of a column of data</td>
<td>SQLGetData Function</td>
</tr>
<tr>
<td>Preparing an SQL statement for execution</td>
<td>SQLPrepare Function</td>
</tr>
<tr>
<td>Setting cursor scrolling options</td>
<td>SQLSetStmtAttr Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft
SQLParamData Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLParamData is used together with SQLPutData to supply parameter data at statement execution time, and with SQLGetData to retrieve streamed output parameter data.

Syntax

```c
SQLRETURN SQLParamData(
    SQLHSTMT StatementHandle,
    SQLPOINTER * ValuePtrPtr);
```

Arguments

**StatementHandle**
[Input] Statement handle.

**ValuePtrPtr**
[Output] Pointer to a buffer in which to return the address of the ParameterValuePtr buffer specified in SQLBindParameter (for parameter data) or the address of the TargetValuePtr buffer specified in SQLBindCol (for column data), as contained in the SQL_DESC_DATA_PTR descriptor record field.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_NO_DATA, SQL_STILL_EXECUTING, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_PARAM_DATA_AVAILABLE.

Diagnostics

When SQLParamData returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values typically returned by SQLParamData and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.
<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>The data value identified by the <code>ValueType</code> argument in <code>SQLBindParameter</code> for the bound parameter could not be converted to the data type identified by the <code>ParameterType</code> argument in <code>SQLBindParameter</code>. The data value returned for a parameter bound as SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT could not be converted to the data type identified by the <code>ValueType</code> argument in <code>SQLBindParameter</code>. (If the data values for one or more rows could not be converted, but one or more rows were successfully returned, this function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>22026</td>
<td>String data, length mismatch</td>
<td>The <code>SQL_NEED_LONG_DATA_LEN</code> information type in <code>SQLGetInfo</code> was &quot;Y&quot;, and less data was sent for a long parameter (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long data source–specific data type) than was specified with the <code>StrLen_or_IndPtr</code> argument in <code>SQLBindParameter</code>. The <code>SQL_NEED_LONG_DATA_LEN</code> information type in <code>SQLGetInfo</code> was &quot;Y&quot;, and less data was sent for a long column (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long data source–specific data type) than was specified in the length buffer corresponding to a column in a row of data that was added or updated with <code>SQLBulkOperations</code> or updated with <code>SQLSetPos</code>.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back because of a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <code>*MessageText</code> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the</td>
</tr>
</tbody>
</table>
The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle; the function was then called again on the StatementHandle.

The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle from a different thread in a multithread application.

| HY010 | Function sequence error | (DM) The previous function call was not a call to SQLExecDirect, SQLExecute, SQLBulkOperations, or SQLSetPos where the return code was SQL_NEED_DATA, or the previous function call was a call to SQLPutData. The previous function call was a call to SQLParamData. (DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLParamData function was called. (DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called. SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. SQLCancel was called before data was sent for all data-at-execution parameters or columns. |
| HY013 | Memory management error | The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions. |
| HY117 | Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed. | (DM) For more information about suspended state, see SQLEndTran Function. |
| HYT01 | Connection timeout expired | The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT. |
| IM001 | Driver does not support this function | (DM) The driver that corresponds to the StatementHandle does not support the function. |
| IM017 | Polling is disabled in asynchronous notification mode | Whenever the notification model is used, polling is disabled. |
| IM018 | SQLCompleteAsync | If the previous function call on the handle returns |
has not been called to complete the previous asynchronous operation on this handle.

SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.

If SQLParamData is called while sending data for a parameter in an SQL statement, it can return any SQLSTATE that can be returned by the function called to execute the statement (SQLExecute or SQLExecDirect). If it is called while sending data for a column being updated or added with SQLBulkOperations or being updated with SQLSetPos, it can return any SQLSTATE that can be returned by SQLBulkOperations or SQLSetPos.

Comments

SQLParamData can be called to supply data-at-execution data for two uses: parameter data that will be used in a call to SQLExecute or SQLExecDirect, or column data that will be used when a row is updated or added by a call to SQLBulkOperations or updated by a call to SQLSetPos. At execution time, SQLParamData returns to the application an indicator of which data the driver requires.

When an application calls SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos, the driver returns SQL_NEED_DATA if it needs data-at-execution data. An application then calls SQLParamData to determine which data to send. If the driver requires parameter data, the driver returns in the *ValuePtrPtr output buffer the value that the application put in the rowset buffer. The application can use this value to determine which parameter data the driver is requesting. If the driver requires column data, the driver returns in the *ValuePtrPtr buffer the address that the column was originally bound to, as follows:

\[
\text{Bound Address} + \text{Binding Offset} + ((\text{Row Number} - 1) \times \text{Element Size})
\]

where the variables are defined as indicated in the following table.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bound Address</td>
<td>The address specified with the TargetValuePtr argument in SQLBindCol.</td>
</tr>
<tr>
<td>Binding Offset</td>
<td>The value stored at the address specified with the SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute.</td>
</tr>
<tr>
<td>Row Number</td>
<td>The 1-based number of the row in the rowset. For single-row fetches, which are the default, this is 1.</td>
</tr>
<tr>
<td>Element Size</td>
<td>The value of the SQL_ATTR_ROW_BIND_TYPE statement attribute for both data and length/indicator buffers.</td>
</tr>
</tbody>
</table>

It also returns SQL_NEED_DATA, which is an indicator to the application that it should call SQLPutData to send the data.

The application calls SQLPutData as many times as necessary to send the data-at-execution data for the column or parameter. After all the data has been sent for the column or parameter, the application
calls SQLParamData again. If SQLParamData again returns SQL_NEED_DATA, data must be sent for another parameter or column. Therefore, the application again calls SQLPutData. If all data-at-execution data has been sent for all parameters or columns, then SQLParamData returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the value in *ValuePtrPtr is undefined, and the SQL statement can be executed or the SQLBulkOperations or SQLSetPos call can be processed.

If SQLParamData supplies parameter data for a searched update or delete statement that does not affect any rows at the data source, the call to SQLParamData returns SQL_NO_DATA.

For more information about how data-at-execution parameter data is passed at statement execution time, see "Passing Parameter Values" in SQLBindParameter and Sending Long Data. For more information about how data-at-execution column data is updated or added, see the section "Using SQLSetPos" in SQLSetPos, "Performing Bulk Updates Using Bookmarks" in SQLBulkOperations, and Long Data and SQLSetPos and SQLBulkOperations.

SQLParamData can be called to retrieve streamed output parameters. When SQLMoreResults, SQLExecute, SQLGetData, or SQLExecDirect returns SQL_PARAM_DATA_AVAILABLE, call SQLParamData to determine which parameter has a value available. For more information about SQL_PARAM_DATA_AVAILABLE and streamed output parameters, see Retrieving Output Parameters Using SQLGetData.

Code Example

See SQLPutData.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a parameter</td>
<td>SQLBindParameter Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Returning information about a parameter in a statement</td>
<td>SQLDescribeParam Function</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
<tr>
<td>Sending parameter data at execution time</td>
<td>SQLPutData Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files
Retrieving Output Parameters Using SQLGetData
SQLParamOptions Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

Summary
The ODBC 2.0 function SQLParamOptions has been replaced in ODBC 3.x by calls to SQLSetStmtAttr.

Note
For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.

Remarks
See ODBC 64-Bit Information, if your application will run on a 64-bit operating system.

See Also
ODBC API Reference
ODBC Header Files

SQLPrepare Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLPrepare prepares an SQL string for execution.

Syntax

```c
SQLRETURN SQLPrepare(
    SQLHSTMT StatementHandle,
```

© 2016 Microsoft
Arguments

StatementHandle
[Input] Statement handle.

StatementText
[Input] SQL text string.

TextLength
[Input] Length of *StatementText in characters.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLPrepare returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLPrepare and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed A specified statement attribute was invalid because of implementation working conditions, so a similar value was temporarily substituted. (SQLGetStmtAttr can be called to determine what the temporarily substituted value is.) The substitute value is valid for the StatementHandle until the cursor is closed. The statement attributes that can be changed are: SQL_ATTR_CONCURRENCY SQL_ATTR_CURSOR_TYPE SQL_ATTR_KEYSET_SIZE SQL_ATTR_MAX_LENGTH SQL_ATTR_MAX_ROWS SQL_ATTR_QUERY_TIMEOUT SQL_ATTR_SIMULATE_CURSOR (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
</tr>
<tr>
<td>21S01</td>
<td>Insert value list does not match column list</td>
</tr>
<tr>
<td>21S02</td>
<td>Degree of derived table does not match column list</td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast specification</td>
</tr>
<tr>
<td>22019</td>
<td>Invalid escape character</td>
</tr>
<tr>
<td>22025</td>
<td>Invalid escape sequence</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>34000</td>
<td>Invalid cursor name</td>
</tr>
<tr>
<td>3D000</td>
<td>Invalid catalog name</td>
</tr>
<tr>
<td>3F000</td>
<td>Invalid schema name</td>
</tr>
<tr>
<td>42000</td>
<td>Syntax error or access violation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>42S01</td>
<td>Base table or view already exists</td>
</tr>
<tr>
<td>42S02</td>
<td>Base table or view not found</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
the specified table name did not exist.

*StatementText* contained a **CREATE VIEW** statement, and a table name or view name defined by the query specification did not exist.

*StatementText* contained a **CREATE INDEX** statement, and the specified table name did not exist.

*StatementText* contained a **GRANT** or **REVOKE** statement, and the specified table name or view name did not exist.

*StatementText* contained a **SELECT** statement, and a specified table name or view name did not exist.

*StatementText* contained a **DELETE, INSERT, or UPDATE** statement, and the specified table name did not exist.

*StatementText* contained a **CREATE TABLE** statement, and a table specified in a constraint (referencing a table other than the one being created) did not exist.

<table>
<thead>
<tr>
<th>Code</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>42S11</td>
<td>Index already exists</td>
</tr>
<tr>
<td></td>
<td><em>StatementText</em> contained a <strong>CREATE INDEX</strong> statement, and the specified index name already existed.</td>
</tr>
<tr>
<td>42S12</td>
<td>Index not found</td>
</tr>
<tr>
<td></td>
<td><em>StatementText</em> contained a <strong>DROP INDEX</strong> statement, and the specified index name did not exist.</td>
</tr>
<tr>
<td>42S21</td>
<td>Column already exists</td>
</tr>
<tr>
<td></td>
<td><em>StatementText</em> contained an <strong>ALTER TABLE</strong> statement, and the column specified in the <strong>ADD</strong> clause is not unique or identifies an existing column in the base table.</td>
</tr>
<tr>
<td>42S22</td>
<td>Column not found</td>
</tr>
<tr>
<td></td>
<td><em>StatementText</em> contained a <strong>CREATE INDEX</strong> statement, and one or more of the column names specified in the column list did not exist.</td>
</tr>
<tr>
<td></td>
<td><em>StatementText</em> contained a <strong>GRANT</strong> or <strong>REVOKE</strong> statement, and a specified column name did not exist.</td>
</tr>
<tr>
<td></td>
<td><em>StatementText</em> contained a <strong>SELECT, DELETE, INSERT, or UPDATE</strong> statement, and a specified column name did not exist.</td>
</tr>
<tr>
<td></td>
<td><em>StatementText</em> contained a <strong>CREATE TABLE</strong> statement, and a column specified in a constraint (referencing a table other than the one being created) did not exist.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
</tr>
<tr>
<td></td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <strong>SQLGetDiagRec</strong> in the <strong>MessageText</strong> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
</tr>
<tr>
<td></td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
</tr>
<tr>
<td>IM018</td>
<td>SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.</td>
</tr>
</tbody>
</table>

**Comments**

The application calls **SQLPrepare** to send an SQL statement to the data source for preparation. For more information about prepared execution, see Prepared Execution. The application can include one or more parameter markers in the SQL statement. To include a parameter marker, the application embeds a question mark (?) into the SQL string at the appropriate position. For information about parameters, see Statement Parameters.

**Note**

If an application uses **SQLPrepare** to prepare and **SQLExecute** to submit a COMMIT or ROLLBACK statement, it will not be interoperable between DBMS products. To commit or roll back a transaction, call **SQLEndTran**.

The driver can modify the statement to use the form of SQL used by the data source and then submit it to the data source for preparation. In particular, the driver modifies the escape sequences used to define SQL syntax for certain features. (For a description of SQL statement grammar, see Escape Sequences in ODBC and Appendix C: SQL Grammar.) For the driver, a statement handle is similar to a statement identifier in embedded SQL code. If the data source supports statement identifiers, the driver can send a statement identifier and parameter values to the data source.

After a statement is prepared, the application uses the statement handle to refer to the statement in later function calls. The prepared statement associated with the statement handle can be re-executed by calling **SQLExecute** until the application frees the statement with a call to **SQLFreeStmt** with the SQL_DROP option or until the statement handle is used in a call to **SQLPrepare**, **SQLExecDirect**, or...
one of the catalog functions (SQLColumns, SQLTables, and so on). Once the application prepares a statement, it can request information about the format of the result set. For some implementations, calling SQLDescribeCol or SQLDescribeParam after SQLPrepare might not be as efficient as calling the function after SQLExecute or SQLEndTran.

Some drivers cannot return syntax errors or access violations when the application calls SQLPrepare. A driver can handle syntax errors and access violations, only syntax errors, or neither syntax errors nor access violations. Therefore, an application must be able to handle these conditions when calling subsequent related functions such as SQLNumResultCols, SQLDescribeCol, SQLColAttribute, and SQLExecute.

Depending on the capabilities of the driver and data source, parameter information (such as data types) might be checked when the statement is prepared (if all parameters have been bound) or when it is executed (if all parameters have not been bound). For maximum interoperability, an application should unbind all parameters that applied to an old SQL statement before preparing a new SQL statement on the same statement. This prevents errors that are due to old parameter information being applied to the new statement.

**Important**

Committing a transaction, either by explicitly calling SQLEndTran or by working in autocommit mode, can cause the data source to delete the access plans for all statements on a connection. For more information, see the SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR information types in SQLGetInfo and Effect of Transactions on Cursors and Prepared Statements.

**Code Example**

See SQLBindParameter, SQLPutData, and SQLSetPos.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocating a statement handle</td>
<td>SQLAllocHandle Function</td>
</tr>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Binding a buffer to a parameter</td>
<td>SQLBindParameter Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Executing a commit or rollback operation</td>
<td>SQLEndTran Function</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
<tr>
<td>Returning the number of rows affected by a statement</td>
<td>SQLRowCount Function</td>
</tr>
</tbody>
</table>
SQLPrimaryKeys Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ODBC

Summary
SQLPrimaryKeys returns the column names that make up the primary key for a table. The driver returns the information as a result set. This function does not support returning primary keys from multiple tables in a single call.

Syntax

```
SQLRETURN SQLPrimaryKeys(
    SQLHSTMT StatementHandle,
    SQLCHAR * CatalogName,
    SQLSMALLINT NameLength1,
    SQLCHAR * SchemaName,
    SQLSMALLINT NameLength2,
    SQLCHAR * TableName,
    SQLSMALLINT NameLength3);
```

Arguments

StatementHandle
[Input] Statement handle.

CatalogName
[Input] Catalog name. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have catalogs. CatalogName cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, CatalogName is treated as an identifier and its case is not significant. If it is SQL_FALSE, CatalogName is an ordinary
argument; it is treated literally, and its case is significant. For more information, see Arguments in Catalog Functions.

**NameLength1**

[Input] Length in characters of *CatalogName*.

**SchemaName**

[Input] Schema name. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string (""") denotes those tables that do not have schemas. *SchemaName* cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *SchemaName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *SchemaName* is an ordinary argument; it is treated literally, and its case is not significant.

**NameLength2**

[Input] Length in characters of *SchemaName*.

**TableName**

[Input] Table name. This argument cannot be a null pointer. *TableName* cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *TableName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *TableName* is an ordinary argument; it is treated literally, and its case is not significant.

**NameLength3**

[Input] Length in characters of *TableName*.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When SQLPrimaryKeys returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLPrimaryKeys and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state (DM) A cursor was open on the <code>StatementHandle</code>, and <code>SQLFetch</code> or <code>SQLFetchScroll</code> had been called.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A cursor was open on the <code>StatementHandle</code>, but <code>SQLFetch</code> or <code>SQLFetchScroll</code> had not been called.</td>
<td></td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The transaction was rolled back due to a resource deadlock with another transaction.</td>
<td></td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <code>*MessageText</code> buffer describes the error and its cause.</td>
<td></td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
<td></td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Asynchronous processing was enabled for the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code>. Then the function was called again on the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code> from a different thread in a multithread application.</td>
<td></td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer (DM) The <code>TableName</code> argument was a null pointer.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the <code>CatalogName</code> argument was a null pointer, and <code>SQLGetInfo</code> with the SQL_CATALOG_NAME information type returns that catalog names are supported. (DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the <code>SchemaName</code> argument was a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error (DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when the <code>SQLPrimaryKeys</code> function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for the <code>StatementHandle</code> and returned</td>
<td></td>
</tr>
</tbody>
</table>
SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.

(DM) An asynchronously executing function (not this one) was called for the `StatementHandle` and was still executing when this function was called.

(DM) `SQLExecute`, `SQLExecDirect`, `SQLBulkOperations`, or `SQLSetPos` was called for the `StatementHandle` and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value of one of the name length arguments was less than 0 but not equal to SQL_NTS, and the associated name argument is not a null pointer. The value of one of the name length arguments exceeded the maximum length value for the corresponding name.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see <code>SQLEndTran Function</code>.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>A catalog was specified, and the driver or data source does not support catalogs. A schema was specified and the driver or data source does not support schemas. The combination of the current settings of the <code>SQL_ATTR_CONCURRENCY</code> and <code>SQL_ATTR_CURSOR_TYPE</code> statement attributes was not supported by the driver or data source. The <code>SQL_ATTR_USE_BOOKMARKS</code> statement attribute was set to SQL_UB_VARIABLE, and the <code>SQL_ATTR_CURSOR_TYPE</code> statement attribute was set to a cursor type for which the driver does not support bookmarks.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The timeout period expired before the data source returned the requested result set. The timeout period is set through <code>SQLSetStmtAttr</code>, <code>SQL_ATTR_QUERY_TIMEOUT</code>.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout...</td>
</tr>
</tbody>
</table>
period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.

| IM001 | Driver does not support this function | (DM) The driver associated with the StatementHandle does not support the function. |
| IM017 | Polling is disabled in asynchronous notification mode | Whenever the notification model is used, polling is disabled. |
| IM018 | SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle. | If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation. |

**Comments**

**SQLPrimaryKeys** returns the results as a standard result set, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and KEY_SEQ. For information about how this information might be used, see Uses of Catalog Data.

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>TABLE_CAT</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>TABLE_SCHEM</td>
</tr>
</tbody>
</table>

To determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns, call SQLGetInfo with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

**Note**

For more information about the general use, arguments, and returned data of ODBC catalog functions, see Catalog Functions.

The following table lists the columns in the result set. Additional columns beyond column 6 (PK_NAME) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set rather than specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_CAT</td>
<td>1</td>
<td>Varchar</td>
<td>Primary key table catalog name; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;&quot;) for those tables that do not have catalogs.</td>
</tr>
<tr>
<td>TABLE_SCHEM</td>
<td>2</td>
<td>Varchar</td>
<td>Primary key table schema name; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;&quot;) for those tables that do not have schemas.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>3</td>
<td>Varchar not NULL</td>
<td>Primary key table name.</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>4</td>
<td>Varchar not NULL</td>
<td>Primary key column name. The driver returns an empty string for a column that does not have a name.</td>
</tr>
<tr>
<td>KEY_SEQ</td>
<td>5</td>
<td>Smallint not NULL</td>
<td>Column sequence number in key (starting with 1).</td>
</tr>
<tr>
<td>PK_NAME</td>
<td>6</td>
<td>Varchar</td>
<td>Primary key name. NULL if not applicable to the data source.</td>
</tr>
</tbody>
</table>

**Code Example**

See [SQLForeignKeys](#).

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Returning the columns of foreign keys</td>
<td>SQLForeignKeys Function</td>
</tr>
<tr>
<td>Returning table statistics and indexes</td>
<td>SQLStatistics Function</td>
</tr>
</tbody>
</table>
SQLProcedureColumns Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ODBC

Summary
SQLProcedureColumns returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures. The driver returns the information as a result set on the specified statement.

Syntax

```c
SQLRETURN SQLProcedureColumns(
    SQLHSTMT StatementHandle,
    SQLCHAR * CatalogName,
    SQLSMALLINT NameLength1,
    SQLCHAR * SchemaName,
    SQLSMALLINT NameLength2,
    SQLCHAR * ProcName,
    SQLSMALLINT NameLength3,
    SQLCHAR * ColumnName,
    SQLSMALLINT NameLength4);
```

Arguments

**StatementHandle**
[Input] Statement handle.

**CatalogName**
[Input] Procedure catalog name. If a driver supports catalogs for some procedures but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those procedures that do not have catalogs. CatalogName cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, CatalogName is treated as an identifier and its case is not significant. If it is SQL_FALSE, CatalogName is an ordinary argument; it is treated literally, and its case is significant. For more information, see Arguments
in Catalog Functions.

**NameLength1**

[Input] Length in characters of *CatalogName.

**SchemaName**

[Input] String search pattern for procedure schema names. If a driver supports schemas for some procedures but not for others, such as when the driver retrieves data from different DBMSs, an empty string (""") denotes those procedures that do not have schemas.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, SchemaName is treated as an identifier and its case is not significant. If it is SQL_FALSE, SchemaName is a pattern value argument; it is treated literally, and its case is significant.

**NameLength2**

[Input] Length in characters of *SchemaName.

**ProcName**

[Input] String search pattern for procedure names.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, ProcName is treated as an identifier and its case is not significant. If it is SQL_FALSE, ProcName is a pattern value argument; it is treated literally, and its case is significant.

**NameLength3**

[Input] Length in characters of *ProcName.

**ColumnName**

[Input] String search pattern for column names.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, ColumnName is treated as an identifier and its case is not significant. If it is SQL_FALSE, ColumnName is a pattern value argument; it is treated literally, and its case is significant.

**NameLength4**

[Input] Length in characters of *ColumnName.

**Returns**

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

When **SQLProcedureColumns** returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling **SQLGetDiagRec** with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by **SQLProcedureColumns** and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.
<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>A cursor was open on the <code>StatementHandle</code>, and <code>SQLFetch</code> or <code>SQLFetchScroll</code> had been called. This error is returned by the Driver Manager if <code>SQLFetch</code> or <code>SQLFetchScroll</code> has not returned SQL_NO_DATA, and is returned by the driver if <code>SQLFetch</code> or <code>SQLFetchScroll</code> has returned SQL_NO_DATA. A cursor was open on the <code>StatementHandle</code>, but <code>SQLFetch</code> or <code>SQLFetchScroll</code> had not been called.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back due to a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLERROR in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code>. Then the function was called again on the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the CatalogName argument was a null pointer, and the SQL_CATALOG_NAME InfoType returns that catalog names are supported. (DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the SchemaName, ProcName, or ColumnName argument was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence</td>
<td>(DM) An asynchronously executing function was called for</td>
</tr>
</tbody>
</table>
the connection handle that is associated with the `StatementHandle`. This asynchronous function was still executing when the SQLProcedureColumns function was called.

(DM) **SQLExecute, SQLExecDirect, or SQLMoreResults** was called for the `StatementHandle` and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.

(DM) An asynchronously executing function (not this one) was called for the `StatementHandle` and was still executing when this function was called.

(DM) **SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos** was called for the `StatementHandle` and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value of one of the name length arguments was less than 0 but not equal to SQL_NTS. The value of one of the name length arguments exceeded the maximum length value for the corresponding catalog, schema, procedure, or column name.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see <code>SQLEndTran Function</code>.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>A procedure catalog was specified, and the driver or data source does not support catalogs. A procedure schema was specified, and the driver or data source does not support schemas. A string search pattern was specified for the procedure schema, procedure name, or column name, and the data source does not support search patterns for one or more of those arguments. The combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes was not supported by the driver or data source. The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks.</td>
</tr>
</tbody>
</table>

---

Note: The details for each error code are based on the provided context and may not fully replicate the exact error messages or details in a real-world application.
HYT00  |  Timeout expired  |  The timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.

HYT01  |  Connection timeout expired  |  The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.

IM001  |  Driver does not support this function  |  (DM) The driver associated with the StatementHandle does not support the function.

IM017  |  Polling is disabled in asynchronous notification mode  |  Whenever the notification model is used, polling is disabled.

IM018  |  SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.  |  If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.

Comments

This function is typically used before statement execution to retrieve information about procedure parameters and the columns that make up the result set or sets returned by the procedure, if any. For more information, see Procedures.

Note

SQLProcedureColumns might not return all columns used by a procedure. For example, a driver might return only information about the parameters used by a procedure and not the columns in a result set it generates.

The SchemaName, ProcName, and ColumnName arguments accept search patterns. For more information about valid search patterns, see Pattern Value Arguments.

Note

For more information about the general use, arguments, and returned data of ODBC catalog functions, see Catalog Functions.

SQLProcedureColumns returns the results as a standard result set, ordered by PROCEDURE_CAT, PROCEDURE_SCHEM, PROCEDURE_NAME, and COLUMN_TYPE. Column names are returned for each procedure in the following order: the name of the return value, the names of each parameter in the procedure invocation (in call order), and then the names of each column in the result set returned by the procedure (in column order).

Applications should bind driver-specific columns relative to the end of the result set. For more information, see Data Returned by Catalog Functions.
To determine the actual lengths of the PROCEDURE_CAT, PROCEDURE_SCHEM, PROCEDURE_NAME, and COLUMN_NAME columns, an application can call `SQLGetInfo` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_PROCEDURE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE_QUALIFIER</td>
<td>PROCEDURE_CAT</td>
</tr>
<tr>
<td>PROCEDURE_OWNER</td>
<td>PROCEDURE_SCHEM</td>
</tr>
<tr>
<td>PRECISION</td>
<td>COLUMN_SIZE</td>
</tr>
<tr>
<td>LENGTH</td>
<td>BUFFER_LENGTH</td>
</tr>
<tr>
<td>SCALE</td>
<td>DECIMAL_DIGITS</td>
</tr>
<tr>
<td>RADIX</td>
<td>NUMPREC_RADIX</td>
</tr>
</tbody>
</table>

The following columns have been added to the results set returned by `SQLProcedureColumns` for ODBC 3.x:

- COLUMN_DEF
- DATETIME_CODE
- CHAR_OCTET_LENGTH
- ORDINAL_POSITION
- IS_NULLABLE

The following table lists the columns in the result set. Additional columns beyond column 19 (IS_NULLABLE) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set rather than specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
</table>
| PROCEDURE_CAT (ODBC 2.0)     | 1             | Varchar   | Procedure catalog name; NULL if not applicable to the data source. If a driver supports catalogs for some procedures but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (“”) for those procedures that do not have catalogs.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type Code</th>
<th>Type Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE_SCHEM (ODBC 2.0)</td>
<td>2</td>
<td>Varchar</td>
<td>Procedure schema name; NULL if not applicable to the data source. If a driver supports schemas for some procedures but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;&quot;) for those procedures that do not have schemas.</td>
</tr>
<tr>
<td>PROCEDURE_NAME (ODBC 2.0)</td>
<td>3</td>
<td>Varchar</td>
<td>Procedure name. An empty string is returned for a procedure that does not have a name.</td>
</tr>
<tr>
<td>COLUMN_NAME (ODBC 2.0)</td>
<td>4</td>
<td>Varchar</td>
<td>Procedure column name. The driver returns an empty string for a procedure column that does not have a name.</td>
</tr>
<tr>
<td>COLUMN_TYPE (ODBC 2.0)</td>
<td>5</td>
<td>Smallint</td>
<td>Defines the procedure column as a parameter or a result set column:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SQL_PARAM_TYPE_UNKNOWN: The procedure column is a parameter whose type is unknown. (ODBC 1.0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SQL_PARAM_INPUT: The procedure column is an input parameter. (ODBC 1.0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SQL_PARAM_INPUT_OUTPUT: The procedure column is an input/output parameter. (ODBC 1.0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SQL_PARAM_OUTPUT: The procedure column is an output parameter. (ODBC 2.0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SQL_RETURN_VALUE: The procedure column is the return value of the procedure. (ODBC 2.0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SQL_RESULT_COL: The procedure column is a result set column. (ODBC 1.0)</td>
</tr>
<tr>
<td>DATA_TYPE (ODBC 2.0)</td>
<td>6</td>
<td>Smallint</td>
<td>SQL data type. This can be an ODBC SQL data type or a driver-specific SQL data type. For datetime and interval data types, this column returns the concise data types (for example, SQL_TYPE_TIME or SQL_INTERVAL_YEAR_TO_MONTH). For a list of valid ODBC SQL data types, see SQL Data Types in Appendix D: Data Types. For information about driver-specific SQL data types, see the driver’s documentation.</td>
</tr>
<tr>
<td>TYPE_NAME (ODBC 2.0)</td>
<td>7</td>
<td>Varchar</td>
<td>Data source-dependent data type name; for example, &quot;CHAR&quot;, &quot;VARCHAR&quot;, &quot;MONEY&quot;, &quot;LONG VARBINARY&quot;, or &quot;CHAR ( ) FOR BIT DATA&quot;.</td>
</tr>
<tr>
<td>COLUMN_SIZE (ODBC 2.0)</td>
<td>8</td>
<td>Integer</td>
<td>The column size of the procedure column on the data source. NULL is returned for data types where column size is not applicable. For more information concerning precision, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>--------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>BUFFER_LENGTH</td>
<td>9</td>
<td>Integer The length in bytes of data transferred on an SQLGetData or SQLFetch operation if SQL_C_DEFAULT is specified. For numeric data, this size may be different than the size of the data stored on the data source. For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size, in Appendix D: Data Types.</td>
<td></td>
</tr>
<tr>
<td>DECIMAL_DIGITS</td>
<td>10</td>
<td>Smallint The decimal digits of the procedure column on the data source. NULL is returned for data types where decimal digits is not applicable. For more information concerning decimal digits, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size, in Appendix D: Data Types.</td>
<td></td>
</tr>
<tr>
<td>NUM_PREC_RADIX</td>
<td>11</td>
<td>Smallint For numeric data types, either 10 or 2. If 10, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of decimal digits allowed for the column. For example, a DECIMAL(12,5) column would return a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 12, and a DECIMAL_DIGITS of 5; a FLOAT column could return a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 15, and a DECIMAL_DIGITS of NULL. If 2, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of bits allowed in the column. For example, a FLOAT column could return a NUM_PREC_RADIX of 2, a COLUMN_SIZE of 53, and a DECIMAL_DIGITS of NULL. NULL is returned for data types where NUM_PREC_RADIX is not applicable.</td>
<td></td>
</tr>
<tr>
<td>NULLABLE</td>
<td>12</td>
<td>Smallint not NULL Whether the procedure column accepts a NULL value: SQL_NO_NULLS: The procedure column does not accept NULL values. SQL_NULLABLE: The procedure column accepts NULL values. SQL_NULLABLEUNKNOWN: It is not known if the procedure column accepts NULL values.</td>
<td></td>
</tr>
<tr>
<td>REMARKS</td>
<td>13</td>
<td>Varchar A description of the procedure column.</td>
<td></td>
</tr>
<tr>
<td>COLUMN_DEF</td>
<td>14</td>
<td>Varchar The default value of the column. If NULL was specified as the default value, this</td>
<td></td>
</tr>
</tbody>
</table>
column is the word NULL, not enclosed in quotation marks. If the default value cannot be represented without truncation, this column contains TRUNCATED, with no enclosing single quotation marks. If no default value was specified, this column is NULL.

The value of COLUMN_DEF can be used in generating a new column definition, except when it contains the value TRUNCATED.

<table>
<thead>
<tr>
<th>SQL_DATA_TYPE (ODBC 3.0)</th>
<th>15</th>
<th>Smallint not NULL</th>
</tr>
</thead>
</table>
| The value of the SQL data type as it appears in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column, except for datetime and interval data types.

For datetime and interval data types, the SQL_DATA_TYPE field in the result set will return SQL_INTERVAL or SQL_DATETIME, and the SQL_DATETIME_SUB field will return the subcode for the specific interval or datetime data type. (See Appendix D: Data Types.) |

<table>
<thead>
<tr>
<th>SQL_DATETIME_SUB (ODBC 3.0)</th>
<th>16</th>
<th>Smallint</th>
</tr>
</thead>
<tbody>
<tr>
<td>The subtype code for datetime and interval data types. For other data types, this column returns a NULL.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAR_OCTET_LENGTH (ODBC 3.0)</th>
<th>17</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>The maximum length in bytes of a character or binary data type column. For all other data types, this column returns a NULL.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ORDINAL_POSITION (ODBC 3.0)</th>
<th>18</th>
<th>Integer not NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>For input and output parameters, the ordinal position of the parameter in the procedure definition (in increasing parameter order, starting at 1). For a return value (if any), 0 is returned. For result-set columns, the ordinal position of the column in the result set, with the first column in the result set being number 1. If there are multiple result sets, column ordinal positions are returned in a driver-specific manner.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IS_NULLABLE (ODBC 3.0)</th>
<th>19</th>
<th>Varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;NO&quot; if the column does not include NULLs. &quot;YES&quot; if the column can include NULLs. This column returns a zero-length string if nullability is unknown. ISO rules are followed to determine nullability. An ISO SQL-compliant DBMS cannot return an empty string. The value returned for this column is different from the value returned for the NULLABLE column. (See the description of the NULLABLE column.)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Code Example

See Procedure Calls.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Returning a list of procedures in a data source</td>
<td>SQLProcedures Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLProcedures Function

Conformance

Version Introduced: ODBC 1.0 Standards Compliance: ODBC

Summary

SQLProcedures returns the list of procedure names stored in a specific data source. Procedure is a generic term used to describe an executable object, or a named entity that can be invoked using input and output parameters. For more information on procedures, see the Procedures.

Syntax

```c
SQLRETURN SQLProcedures(
    SQLHSTMT StatementHandle,
```
Arguments

StatementHandle
[Input] Statement handle.

CatalogName
[Input] Procedure catalog. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string (""") denotes those tables that do not have catalogs. *CatalogName* cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *CatalogName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *CatalogName* is an ordinary argument; it is treated literally, and its case is significant. For more information, see Arguments in Catalog Functions.

NameLength1
[Input] Length in characters of *CatalogName*.

SchemaName
[Input] String search pattern for procedure schema names. If a driver supports schemas for some procedures but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those procedures that do not have schemas.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *SchemaName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *SchemaName* is a pattern value argument; it is treated literally, and its case is significant.

NameLength2
[Input] Length in characters of *SchemaName*.

ProcName
[Input] String search pattern for procedure names.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *ProcName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *ProcName* is a pattern value argument; it is treated literally, and its case is significant.

NameLength3
[Input] Length in characters of *ProcName*.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.
### Diagnostics

When **SQLProcedures** returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling **SQLGetDiagRec** with a **HandleType** of SQL_HANDLE_STMT and a **Handle** of **StatementHandle**. The following table lists the SQLSTATE values commonly returned by **SQLProcedures** and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>A cursor was open on the <strong>StatementHandle</strong>, and <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong> had been called. This error is returned by the Driver Manager if <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong> has not returned SQL_NO_DATA, and is returned by the driver if <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong> has returned SQL_NO_DATA. A cursor was open on the <strong>StatementHandle</strong>, but <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong> had not been called.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back due to a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <strong>SQLGetDiagRec</strong> in the <em>MessageText</em> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <strong>StatementHandle</strong>. The function was called, and before it completed execution, <strong>SQLCancel</strong> or <strong>SQLCancelHandle</strong> was called on the <strong>StatementHandle</strong>. Then the function was called again on the <strong>StatementHandle</strong>. The function was called, and before it completed execution, <strong>SQLCancel</strong> or <strong>SQLCancelHandle</strong> was called on the <strong>StatementHandle</strong> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Error Message</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the CatalogName argument was a null pointer, and the SQL_CATALOG_NAME InfoType returns that catalog names are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the SchemaName or ProcName argument was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when this function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value of one of the name length arguments was less than 0 but not equal to SQL_NTS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value of one of the name length arguments exceeded the maximum length value for the corresponding name.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>A procedure catalog was specified, and the driver or data source does not support catalogs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A procedure schema was specified, and the driver or data source does not support schemas.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A string search pattern was specified for the procedure schema or procedure name, and the data source does not support schemas.</td>
</tr>
</tbody>
</table>
support search patterns for one or more of those arguments.

The combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes was not supported by the driver or data source.

The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks.

<table>
<thead>
<tr>
<th>HYT00</th>
<th>Timeout expired</th>
<th>The query timeout period expired before the data source returned the requested result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support this function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td>SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

Comments

SQLProcedures lists all procedures in the requested range. A user may or may not have permission to execute any of these procedures. To check accessibility, an application can call SQLGetInfo and check the SQL_ACCESSIBLE_PROCEDURES information value. Otherwise, the application must be able to handle a situation where the user selects a procedure that it cannot execute. For information about how this information might be used, see Procedures.

Note

For more information about the general use, arguments, and returned data of ODBC catalog functions, see Catalog Functions.

SQLProcedures returns the results as a standard result set, ordered by PROCEDURE_CAT, PROCEDURE_SCHEMA, and PROCEDURE_NAME.
**Note**

SQLProcedures might not return all procedures. Applications can use any valid procedure, regardless of whether it is returned by SQLProcedures.

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE_QUALIFIER</td>
<td>PROCEDURE_CAT</td>
</tr>
<tr>
<td>PROCEDURE_OWNER</td>
<td>PROCEDURE_SCHEMA</td>
</tr>
</tbody>
</table>

To determine the actual lengths of the PROCEDURE_CAT, PROCEDURE_SCHEMA, and PROCEDURE_NAME columns, an application can call SQLGetInfo with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, and SQL_MAX_PROCEDURE_NAME_LEN options.

The following table lists the columns in the result set. Additional columns beyond column 8 (PROCEDURE_TYPE) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set rather than specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE_CAT (ODBC 2.0)</td>
<td>1</td>
<td>Varchar</td>
<td>Procedure catalog identifier; NULL if not applicable to the data source. If a driver supports catalogs for some procedures but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those procedures that do not have catalogs.</td>
</tr>
<tr>
<td>PROCEDURE_SCHEMA (ODBC 2.0)</td>
<td>2</td>
<td>Varchar</td>
<td>Procedure schema identifier; NULL if not applicable to the data source. If a driver supports schemas for some procedures but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those procedures that do not have schemas.</td>
</tr>
<tr>
<td>PROCEDURE_NAME (ODBC 2.0)</td>
<td>3</td>
<td>Varchar</td>
<td>Procedure identifier.</td>
</tr>
<tr>
<td>NUM_INPUT_PARAMS (ODBC 2.0)</td>
<td>4</td>
<td>N/A</td>
<td>Reserved for future use. Applications should not rely on the data returned in these result columns.</td>
</tr>
<tr>
<td>NUM_OUTPUT_PARAMS (ODBC 2.0)</td>
<td>5</td>
<td>N/A</td>
<td>Reserved for future use. Applications should not rely on the data returned in these result columns.</td>
</tr>
<tr>
<td>NUM_RESULT_SETS (ODBC 2.0)</td>
<td>6</td>
<td>N/A</td>
<td>Reserved for future use. Applications should not rely on the data returned in these result columns.</td>
</tr>
<tr>
<td>REMARKS (ODBC 2.0)</td>
<td>7</td>
<td>Varchar</td>
<td>A description of the procedure.</td>
</tr>
</tbody>
</table>
| PROCEDURE_TYPE (ODBC 2.0) | 8 | Smallint | Defines the procedure type:  
- SQL_PT_UNKNOWN: It cannot be determined whether the procedure returns a value.  
- SQL_PT_PROCEDURE: The returned object is a procedure; that is, it does not have a return value.  
- SQL_PT_FUNCTION: The returned object is a function; that is, it has a return value. |

The `SchemaName` and `ProcName` arguments accept search patterns. For more information about valid search patterns, see Pattern Value Arguments.

## Code Example

See Procedure Calls.

## Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Returning information about a driver or data source</td>
<td>SQLGetInfo Function</td>
</tr>
<tr>
<td>Returning the parameters and result set columns of a procedure</td>
<td>SQLProcedureColumns Function</td>
</tr>
<tr>
<td>Syntax for invoking stored procedures</td>
<td>Executing Statements</td>
</tr>
</tbody>
</table>

## See Also

ODBC API Reference
SQLPutData Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLPutData allows an application to send data for a parameter or column to the driver at statement execution time. This function can be used to send character or binary data values in parts to a column with a character, binary, or data source–specific data type (for example, parameters of the SQL_LONGVARBINARY or SQL_LONGVARCHAR types). SQLPutData supports binding to a Unicode C data type, even if the underlying driver does not support Unicode data.

Syntax

```c
SQLRETURN SQLPutData(
    SQLHSTMT StatementHandle,
    SQLPOINTER DataPtr,
    SQLLEN StrLen_or_Ind);
```

Arguments

**StatementHandle**
[Input] Statement handle.

**DataPtr**
[Input] Pointer to a buffer containing the actual data for the parameter or column. The data must be in the C data type specified in the ValueType argument of SQLBindParameter (for parameter data) or the TargetType argument of SQLBindCol (for column data).

**StrLen_or_Ind**
[Input] Length of *DataPtr. Specifies the amount of data sent in a call to SQLPutData. The amount of data can vary with each call for a given parameter or column. StrLen_or_Ind is ignored unless it meets one of the following conditions:

- StrLen_or_Ind is SQL_NTS, SQL_NULL_DATA, or SQL_DEFAULT_PARAM.
- The C data type specified in SQLBindParameter or SQLBindCol is SQL_C_CHAR or SQL_C_BINARY.
- The C data type is SQL_C_DEFAULT, and the default C data type for the specified SQL data type is SQL_C_CHAR or SQL_C_BINARY.
For all other types of C data, if StrLen_or_Ind is not SQL_NULL_DATA or SQL_DEFAULT_PARAM, the driver assumes that the size of the *DataPtr buffer is the size of the C data type specified with ValueType or TargetType and sends the entire data value. For more information, see Converting Data from C to SQL Data Types in Appendix D: Data Types.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLPutData returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLPutData and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>String or binary data returned for an output parameter resulted in the truncation of nonblank character or non-NULL binary data. If it was a string value, it was right-truncated. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>The data value identified by the ValueType argument in SQLBindParameter for the bound parameter could not be converted to the data type identified by the ParameterType argument in SQLBindParameter.</td>
</tr>
<tr>
<td>07S01</td>
<td>Invalid use of default parameter</td>
<td>A parameter value, set with SQLBindParameter, was SQL_DEFAULT_PARAM, and the corresponding parameter did not have a default value.</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>22001</td>
<td>String data, right truncation</td>
<td>The assignment of a character or binary value to a column resulted in the truncation of nonblank (character) or non-null (binary) characters or bytes. The SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo was &quot;Y&quot;, and more data was sent for a long...</td>
</tr>
</tbody>
</table>
parameter (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long data source-specific data type) than was specified with the StrLen_or_IndPtr argument in SQLBindParameter.

The SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo was "Y", and more data was sent for a long column (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long data source-specific data type) than was specified in the length buffer corresponding to a column in a row of data that was added or updated with SQLBulkOperations or updated with SQLSetPos.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>22003</td>
<td>Numeric value out of range</td>
<td>The data sent for a bound numeric parameter or column caused the whole (as opposed to fractional) part of the number to be truncated when assigned to the associated table column. Returning a numeric value (as numeric or string) for one or more input/output or output parameters would have caused the whole (as opposed to fractional) part of the number to be truncated.</td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format</td>
<td>The data sent for a parameter or column that was bound to a date, time, or timestamp structure was, respectively, an invalid date, time, or timestamp. An input/output or output parameter was bound to a date, time, or timestamp C structure, and a value in the returned parameter was, respectively, an invalid date, time, or timestamp. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>22008</td>
<td>Datetime field overflow</td>
<td>A datetime expression computed for an input/output or output parameter resulted in a date, time, or timestamp C structure that was invalid.</td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
<td>An arithmetic expression calculated for an input/output or output parameter resulted in division by zero.</td>
</tr>
<tr>
<td>22015</td>
<td>Interval field overflow</td>
<td>The data sent for an exact numeric or interval column or parameter to an interval SQL data type caused a loss of significant digits. Data was sent for an interval column or parameter with more than one field, was converted to a numeric data type, and had no representation in the numeric data type. The data sent for column or parameter data was assigned to an interval SQL type, and there was no representation of the value of the C type in the interval SQL type. The data sent for an exact numeric or interval C column or parameter to an interval C type caused a loss of significant digits. The data sent for column or parameter data was assigned to</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast specification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The C type was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column or parameter was not a valid literal of the bound C type. The SQL type was an exact or approximate numeric, a datetime, or an interval data type; the C type was SQL_C_CHAR; and the value in the column or parameter was not a valid literal of the bound SQL type.</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the StatementHandle. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle. Then the function was called again on the StatementHandle. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>(DM) The argument DataPtr was a null pointer, and the argument StrLen_or_Ind was not 0, SQL_DEFAULT_PARAM, or SQL_NULL_DATA.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) The previous function call was not a call to SQLPutData or SQLParamData. (DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLPutData function was called. (DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. (DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly</td>
</tr>
<tr>
<td>Code</td>
<td>Message</td>
<td>Details</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>HY019</td>
<td>Non-character and non-binary data sent in pieces</td>
<td><strong>SQLPutData</strong> was called more than once for a parameter or column, and it was not being used to send character C data to a column with a character, binary, or data source–specific data type or to send binary C data to a column with a character, binary, or data source–specific data type.</td>
</tr>
<tr>
<td>HY020</td>
<td>Attempt to concatenate a null value</td>
<td><strong>SQLPutData</strong> was called more than once since the call that returned SQL_NEED_DATA, and in one of those calls, the <em>StrLen_or_Ind</em> argument contained SQL_NULL_DATA or SQL_DEFAULT_PARAM.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>The argument <em>DataPtr</em> was not a null pointer, and the argument <em>StrLen_or_Ind</em> was less than 0 but not equal to SQL_NTS or SQL_NULL_DATA.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see <strong>SQLEndTran Function</strong>.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <strong>SQLSetConnectAttr</strong>, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the <em>StatementHandle</em> does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><strong>SQLCompleteAsync</strong> has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, <strong>SQLCompleteAsync</strong> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

If **SQLPutData** is called while sending data for a parameter in an SQL statement, it can return any SQLSTATE that can be returned by the function called to execute the statement (**SQLExecute** or **SQLExecDirect**). If it is called while sending data for a column being updated or added with **SQLBulkOperations** or being updated with **SQLSetPos**, it can return any SQLSTATE that can be returned by **SQLBulkOperations** or **SQLSetPos**.
Comments

SQLPutData can be called to supply data-at-execution data for two uses: parameter data to be used in a call to SQLExecute or SQLExecDirect, or column data to be used when a row is updated or added by a call to SQLBulkOperations or is updated by a call to SQLSetPos.

When an application calls SQLParamData to determine which data it should send, the driver returns an indicator that the application can use to determine which parameter data to send or where column data can be found. It also returns SQL_NEED_DATA, which is an indicator to the application that it should call SQLPutData to send the data. In the DataPtr argument to SQLPutData, the application passes a pointer to the buffer containing the actual data for the parameter or column.

When the driver returns SQL_SUCCESS for SQLPutData, the application calls SQLParamData again. SQLParamData returns SQL_NEED_DATA if more data needs to be sent, in which case the application calls SQLPutData again. It returns SQL_SUCCESS if all data-at-execution data has been sent. The application then calls SQLParamData again. If the driver returns SQL_NEED_DATA and another indicator in *ValuePtrPtr, it requires data for another parameter or column and SQLPutData is called again. If the driver returns SQL_SUCCESS, then all data-at-execution data has been sent and the SQL statement can be executed or the SQLBulkOperations or SQLSetPos call can be processed.

For more information on how data-at-execution parameter data is passed at statement execution time, see "Passing Parameter Values" in SQLBindParameter and Sending Long Data. For more information on how data-at-execution column data is updated or added, see the section "Using SQLSetPos" in SQLSetPos, "Performing Bulk Updates Using Bookmarks" in SQLBulkOperations, and Long Data and SQLSetPos and SQLBulkOperations.

Note

An application can use SQLPutData to send data in parts only when sending character C data to a column with a character, binary, or data source–specific data type or when sending binary C data to a column with a character, binary, or data source–specific data type. If SQLPutData is called more than once under any other conditions, it returns SQL_ERROR and SQLSTATE HY019 (Non-character and non-binary data sent in pieces).

The following sample assumes a data source name called Test. The associated database should have a table that you can create, as follows:

```
CREATE TABLE temp4 (NAME char(30), AGE int, BIRTHDAY datetime, Memo1 text)
```

```
// SQLPutData.cpp
// compile with: odbc32.lib user32.lib
#include <stdio.h>
#include <windows.h>
#include <sqlext.h>
#include <odbcss.h>

#define TEXTSIZE 12000
#define MAXBUFLEN 256

SQLHENV henv = SQL_NULL_HENV;
```
SQLHDBC hdbc1 = SQL_NULL_HDBC;
SQLHSTMT hstmt1 = SQL_NULL_HSTMT;

void Cleanup() {
    if (hstmt1 != SQL_NULL_HSTMT)
        SQLFreeHandle(SQL_HANDLE_STMT, hstmt1);

    if (hdbc1 != SQL_NULL_HDBC) {
        SQLDisconnect(hdbc1);
        SQLFreeHandle(SQL_HANDLE_DBC, hdbc1);
    }

    if (henv != SQL_NULL_HENV)
        SQLFreeHandle(SQL_HANDLE_ENV, henv);
}

int main() {
    RETCODE retcode;

    // SQLBindParameter variables.
    SQLLEN cbTextSize, lbytes;

    // SQLParamData variable.
    PTR pParmID;

    // SQLPutData variables.
    UCHAR Data[] =
        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
        "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
    SDWORD cbBatch = (SDWORD)sizeof(Data) - 1;

    // Allocate the ODBC environment and save handle.
    retcode = SQLAllocHandle (SQL_HANDLE_ENV, NULL, &henv);
    if ( (retcode != SQL_SUCCESS_WITH_INFO) && (retcode != SQL_SUCCESS)) {
        printf("SQLAllocHandle(Env)	Failed\n\n");
        Cleanup();
        return(9);
    }

    // Notify ODBC that this is an ODBC 3.0 app.
    retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_INTEGER);
    if ( (retcode != SQL_SUCCESS_WITH_INFO) && (retcode != SQL_SUCCESS)) {
        printf("SQLSetEnvAttr(ODBC	version)	Failed\n\n");
        Cleanup();
        return(9);
    }

    // Allocate ODBC connection handle and connect.
    retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc1);
    if ( (retcode != SQL_SUCCESS_WITH_INFO) && (retcode != SQL_SUCCESS)) {
        printf("SQLAllocHandle(DBC)	Failed\n\n");
        Cleanup();
        return(9);
    }

}
printf("SQLAllocHandle(hdbc1) Failed\n\n");
Cleanup();
return(9);
}

// Sample uses Integrated Security, create SQL Server DSN using Windows NT authentication.
retcode = SQLConnect(hdbc1, (UCHAR*)"Test", SQL_NTS, (UCHAR*)"", SQL_NTS, (UCHAR*)"", SQL_NTS);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) ) {
    printf("SQLConnect() Failed\n\n");
    Cleanup();
    return(9);
}

// Allocate statement handle.
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) ) {
    printf("SQLAllocHandle(hstmt1) Failed\n\n");
    Cleanup();
    return(9);
}

// Set parameters based on total data to send.
lbytes = (SDWORD)TEXTSIZE;
cbTextSize = SQL_LEN_DATA_AT_EXEC(lbytes);

// Bind the parameter marker.
retcode = SQLBindParameter (hstmt1, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_LONGVARCHAR, lbytes, 0, (VOID *)1, &cbTextSize);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) ) {
    printf("SQLBindParameter Failed\n\n");
    Cleanup();
    return(9);
}

// Execute the command.
retcode = SQLExecDirect(hstmt1, (UCHAR*)"INSERT INTO temp4 VALUES('Paul Borm', 46,'1950-11-12 00:00:00',?)", SQL_NTS);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_NEED_DATA) && (retcode != SQL_SUCCESS_)
    printf("SQLExecDirect Failed\n\n");
    Cleanup();
    return(9);
}

// Check to see if NEED_DATA; if yes, use SQLPutData.
retcode = SQLParamData(hstmt1, &pParmID);
if (retcode == SQL_NEED_DATA) {
    while (lbytes > cbBatch) {
        SQLPutData(hstmt1, Data, cbBatch);
        lbytes -= cbBatch;
    }
}
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a parameter</td>
<td>SQLBindParameter Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Executing an SQL statement</td>
<td>SQLEncryptDirect Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLEncryptFunction</td>
</tr>
<tr>
<td>Returning the next parameter to send data for</td>
<td>SQLParamData Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLRowCount Function
Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLRowCount returns the number of rows affected by an UPDATE, INSERT, or DELETE statement; an SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or SQL_DELETE_BY_BOOKMARK operation in SQLBulkOperations; or an SQL_UPDATE or SQL_DELETE operation in SQLSetPos.

Syntax

```c
SQLRETURN SQLRowCount(
    SQLHSTMT StatementHandle,
    SQLLEN * RowCountPtr);
```

Arguments

StatementHandle
[Input] Statement handle.

RowCountPtr
[Output] Points to a buffer in which to return a row count. For UPDATE, INSERT, and DELETE statements, for the SQL_ADD, SQL_UPDATE_BY_BOOKMARK, and SQL_DELETE_BY_BOOKMARK operations in SQLBulkOperations, and for the SQL_UPDATE or SQL_DELETE operations in SQLSetPos, the value returned in *RowCountPtr is either the number of rows affected by the request or –1 if the number of affected rows is not available.

When SQLExecute, SQLExecDirect, SQLBulkOperations, SQLSetPos, or SQLMoreResults is called, the SQL_DIA GG_ROW_COUNT field of the diagnostic data structure is set to the row count, and the row count is cached in an implementation-dependent way. SQLRowCount returns the cached row count value. The cached row count value is valid until the statement handle is set back to the prepared or allocated state, the statement is reexecuted, or SQLCloseCursor is called. Note that if a function has been called since the SQL_DIA GG_ROW_COUNT field was set, the value returned by SQLRowCount might be different from the value in the SQL_DIA GG_ROW_COUNT field because the SQL_DIA GG_ROW_COUNT field is reset to 0 by any function call.

For other statements and functions, the driver may define the value returned in *RowCountPtr. For example, some data sources may be able to return the number of rows returned by a SELECT statement or a catalog function before fetching the rows.

Note
Many data sources cannot return the number of rows in a result set before fetching them; for maximum interoperability, applications should not rely on this behavior.

Returns
When `SQLRowCount` returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling `SQLGetDiagRec` with a HandleType of SQL_HANDLE_STMT and a Handle of `StatementHandle`. The following table lists the SQLSTATE values commonly returned by `SQLRowCount` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <em>MessageText</em> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when the <code>SQLRowCount</code> function was called.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed,</td>
</tr>
</tbody>
</table>
Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.

Connection timeout expired

Driver does not support this function

Comments

If the last SQL statement executed on the statement handle was not an UPDATE, INSERT, or DELETE statement or if the Operation argument in the previous call to SQLBulkOperations was not SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or SQL_DELETE_BY_BOOKMARK, or if the Operation argument in the previous call to SQLSetPos was not SQL_UPDATE or SQL_DELETE, the value of *RowCountPtr is driver-defined. For more information, see Determining the Number of Affected Rows.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

SQLSetConnectAttr Function
Conformance
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary
SQLSetConnectAttr sets attributes that govern aspects of connections.

Note
For more information about what the Driver Manager maps this function to when an ODBC 3.x application is working with an ODBC 2.x driver, see Mapping Replacement Functions for Backward Compatibility of Applications.

Syntax

```c
SQLRETURN SQLSetConnectAttr(
    SQLHDBC ConnectionHandle,
    SQLINTEGER Attribute,
    SQLPOINTER ValuePtr,
    SQLINTEGER StringLength);
```

Arguments

**ConnectionHandle**
[Input] Connection handle.

**Attribute**
[Input] Attribute to set, listed in "Comments."

**ValuePtr**
[Input] Pointer to the value to be associated with Attribute. Depending on the value of **Attribute**, **ValuePtr** will be an unsigned integer value or will point to a null-terminated character string. Note that the integral type of the **Attribute** argument may not be fixed length, see the Comments section for detail.

**StringLength**
[Input] If **Attribute** is an ODBC-defined attribute and **ValuePtr** points to a character string or a binary buffer, this argument should be the length of **ValuePtr**. For character string data, this argument should contain the number of bytes in the string.

If **Attribute** is an ODBC-defined attribute and **ValuePtr** is an integer, **StringLength** is ignored.

If **Attribute** is a driver-defined attribute, the application indicates the nature of the attribute to the Driver Manager by setting the **StringLength** argument. **StringLength** can have the following values:

- If **ValuePtr** is a pointer to a character string, then **StringLength** is the length of the string or SQL_NTS.
- If **ValuePtr** is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in **StringLength**. This places a negative value in
If `ValuePtr` is a pointer to a value other than a character string or a binary string, then `StringLength` should have the value SQL_IS_POINTER.

If `ValuePtr` contains a fixed-length value, then `StringLength` is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_STILL_EXECUTING.

Diagnostics

When `SQLSetConnectAttr` returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling `SQLGetDiagRec` with a `HandleType` of SQL_HANDLE_DBC and a `Handle` of `ConnectionHandle`. The following table lists the SQLSTATE values commonly returned by `SQLSetConnectAttr` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

The driver can return SQL_SUCCESS_WITH_INFO to provide information about the result of setting an option.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed</td>
</tr>
<tr>
<td>08002</td>
<td>Connection name in use</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
</tr>
<tr>
<td>25000</td>
<td>Illegal operation</td>
</tr>
</tbody>
</table>
while in a local transaction | enlist in a distributed transaction connection (DTC) by setting the connection attribute SQL_ATTR_ENLIST_IN_DTC.

A connection is already enlisted in a DTC.

A connection has been enlisted in a distributed transaction connection and a local transaction was started by setting SQL_ATTR_AUTOCOMMIT to SQL_AUTOCOMMIT_OFF.

| 3D000 | Invalid catalog name | The Attribute argument was SQL_CURRENT_CATALOG, and the specified catalog name was invalid.

| HY000 | General error | An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.

| HY001 | Memory allocation error | The driver was unable to allocate memory required to support execution or completion of the function.

| HY008 | Operation canceled | Asynchronous processing was enabled for the ConnectionHandle. The SQLSetConnectAttr function was called, and before it completed execution, the SQLCancelHandle function was called on the ConnectionHandle, and then the SQLSetConnectAttr function was called again on the ConnectionHandle.

Or, the SQLSetConnectAttr function was called, and before it completed execution, SQLCancelHandle was called on the ConnectionHandle from a different thread in a multithread application.

| HY009 | Invalid use of null pointer | The Attribute argument identified a connection attribute that required a string value, and the ValuePtr argument was a null pointer.

| HY010 | Function sequence error | (DM) An asynchronously executing function was called for a StatementHandle associated with the ConnectionHandle and was still executing when SQLSetConnectAttr was called.

(DM) An asynchronously executing function (not this one) was called for the ConnectionHandle and was still executing when this function was called.

(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for one of the statement handles associated with the ConnectionHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.

(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for a StatementHandle associated with the ConnectionHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Detailed Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY011</td>
<td>Attribute cannot be set now</td>
<td>The <code>Attribute</code> argument was SQL_ATTR_TXN_ISOLATION, and a transaction was open.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY024</td>
<td>Invalid attribute value</td>
<td>Given the specified <code>Attribute</code> value, an invalid value was specified in <code>ValuePtr</code>. (The Driver Manager returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE or SQL_ATTR_ASYNC_ENABLE. For all other connection and statement attributes, the driver must verify the value specified in <code>ValuePtr</code>.) The <code>Attribute</code> argument was SQL_ATTR_TRACEFILE or SQL_ATTR_TRANSLATE_LIB, and <code>ValuePtr</code> was an empty string.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td><em>(DM)</em> <code>ValuePtr</code> is a character string, and the <code>StringLength</code> argument was less than 0 but was not SQL_NTS.</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute/option identifier</td>
<td><em>(DM)</em> The value specified for the argument <code>Attribute</code> was not valid for the version of ODBC supported by the driver. <em>(DM)</em> The value specified for the argument <code>Attribute</code> was a read-only attribute.</td>
</tr>
<tr>
<td>HY114</td>
<td>Driver does not support connection-level asynchronous function execution</td>
<td><em>(DM)</em> An application attempted to enable asynchronous function execution with SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE for a driver that does not support asynchronous connection operations.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td><em>(DM)</em> For more information about the suspended state, see <code>SQLEndTran</code> Function.</td>
</tr>
<tr>
<td>HY121</td>
<td>Cursor Library and Driver-Aware Pooling cannot be enabled at the same time</td>
<td>For more information, see Driver-Aware Connection Pooling.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The value specified for the argument <code>Attribute</code> was a valid ODBC connection or statement attribute for the version of ODBC supported by the driver but was not supported by the driver.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through <code>SQLSetConnectAttr</code>, <code>SQL_ATTR_CONNECTION_TIMEOUT</code>.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the <code>ConnectionHandle</code> does not support the function.</td>
</tr>
<tr>
<td>IM009</td>
<td>Unable to load translation DLL</td>
<td>The driver was unable to load the translation DLL that was specified for the connection. This error can be returned only when <code>Attribute</code> is <code>SQL_ATTR_TRANSLATE_LIB</code>.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><strong>SQLCompleteAsync</strong> has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns <code>SQL_STILL_EXECUTING</code> and if notification mode is enabled, <code>SQLCompleteAsync</code> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
<tr>
<td>S1118</td>
<td>Driver does not support asynchronous notification</td>
<td><code>SQL_ATTR_ASYNC_DBC_EVENT</code> was set (after the connection was made) but asynchronous notification is not supported by the driver.</td>
</tr>
</tbody>
</table>

When `Attribute` is a statement attribute, `SQLSetConnectAttr` can return any SQLSTATEs returned by `SQLSetStmtAttr`.

**Comments**

For general information about connection attributes, see [Connection Attributes](#).

The currently defined attributes and the version of ODBC in which they were introduced are shown in the table later in this section; it is expected that more attributes will be defined to take advantage of different data sources. A range of attributes is reserved by ODBC; driver developers must reserve values for their own driver-specific use from Open Group.

**Note**

The ability to set statement attributes at the connection level by calling `SQLSetConnectAttr` has been deprecated in ODBC 3.x. ODBC 3.x applications should never set statement attributes at the connection level. ODBC 3.x statement attributes cannot be set at the connection level, with the exception of the `SQL_ATTR_METADATA_ID` and `SQL_ATTR_ASYNC_ENABLE` attributes, which are both connection attributes and statement attributes and can be set at either the connection level or the statement level.
ODBC 3.x drivers need only support this functionality if they should work with ODBC 2.x applications that set ODBC 2.x statement options at the connection level. For more information, see SQLSetConnectOption Mapping in Appendix G: Driver Guidelines for Backward Compatibility.

An application can call SQLSetConnectAttr at any time between the time the connection is allocated and freed. All connection and statement attributes successfully set by the application for the connection persist until SQLFreeHandle is called on the connection. For example, if an application calls SQLSetConnectAttr before connecting to a data source, the attribute persists even if SQLSetConnectAttr fails in the driver when the application connects to the data source; if an application sets a driver-specific attribute, the attribute persists even if the application connects to a different driver on the connection.

Some connection attributes can be set only before a connection has been made; others can be set only after a connection has been made. The following table indicates those connection attributes that must be set either before or after a connection has been made. Either indicates that the attribute can be set either before or after connection.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Set before or after connection?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_ACCESS_MODE</td>
<td>Either[1]</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_DBC_EVENT</td>
<td>Either</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE</td>
<td>Either[4]</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_DBC_PCALLBACK</td>
<td>Either</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_DBC_PCONTEXT</td>
<td>Either</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_ENABLE</td>
<td>Either[2]</td>
</tr>
<tr>
<td>SQL_ATTR_AUTO_IPD</td>
<td>Either</td>
</tr>
<tr>
<td>SQL_ATTR_AUTOCOMMIT</td>
<td>Either[5]</td>
</tr>
<tr>
<td>SQL_ATTR_CONNECTION_DEAD</td>
<td>After</td>
</tr>
<tr>
<td>SQL_ATTR_CONNECTION_TIMEOUT</td>
<td>Either</td>
</tr>
<tr>
<td>SQL_ATTR_CURRENT_CATALOG</td>
<td>Either[1]</td>
</tr>
<tr>
<td>SQL_ATTR_DBC_INFO_TOKEN</td>
<td>After</td>
</tr>
<tr>
<td>SQL_ATTR_ENLIST_IN_DTC</td>
<td>After</td>
</tr>
<tr>
<td>SQL_ATTR_LOGIN_TIMEOUT</td>
<td>Before</td>
</tr>
<tr>
<td>SQL_ATTR_METADATA_ID</td>
<td>Either</td>
</tr>
<tr>
<td>SQL_ATTR_ODBC_CURSORS</td>
<td>Before</td>
</tr>
<tr>
<td>SQL_ATTR_PACKET_SIZE</td>
<td>Before</td>
</tr>
<tr>
<td>Attribute</td>
<td>ValuePtr contents</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SQL_ATTR_ACCESS_MODE (ODBC 1.0)</td>
<td>An SQLINTEGER value. SQL_MODE_READ_ONLY is used by the driver or data source as an indicator that the connection is not required to support SQL statements that cause updates to occur. This mode can be used to optimize locking strategies, transaction management, or other areas as...</td>
</tr>
</tbody>
</table>

[1] SQL_ATTR_ACCESS_MODE and SQL_ATTR_CURRENT_CATALOG can be set before or after connecting, depending on the driver. However, interoperable applications set them before connecting because some drivers do not support changing these after connecting.

[2] SQL_ATTR_ASYNC_ENABLE must be set before there is an active statement.

[3] SQL_ATTR_TXN_ISOLATION can be set only if there are no open transactions on the connection. Some connection attributes support substitution of a similar value if the data source does not support the value specified in *ValuePtr. In such cases, the driver returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed). For example, if Attribute is SQL_ATTR_PACKET_SIZE and *ValuePtr exceeds the maximum packet size, the driver substitutes the maximum size. To determine the substituted value, an application calls SQLGetConnectAttr.

[4] If SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE is set before a connection is open, the Driver Manager will set the driver's attribute when the driver is loaded during a call to SQLBrowseConnect, SQLConnect, or SQLDriverConnect. Before a call to SQLBrowseConnect, SQLConnect, or SQLDriverConnect, the Driver Manager does not know which driver to connect to and does not know whether the driver supports asynchronous connection operations. Therefore, the Driver Manager always returns SQL_SUCCESS. But, in case the driver does not support asynchronous connection operations, the call to SQLBrowseConnect, SQLConnect, or SQLDriverConnect will fail.

[5] When SQL_ATTR_AUTOCOMMIT is set to FALSE, applications should call SQLEndTran(SQL_ROLLBACK) if any API returns SQL_ERROR to ensure transactional consistency.
appropriate to the driver or data source. The driver is not required to prevent such statements from being submitted to the data source. The behavior of the driver and data source when asked to process SQL statements that are not read-only during a read-only connection is implementation-defined. SQL_MODE_READ_WRITE is the default.

<table>
<thead>
<tr>
<th>SQL_ATTR_ASYNC_DBC_EVENT (ODBC 3.8)</th>
<th>A SQLPOINTER value that is an event handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Notification of the completion of asynchronous functions is enabled by calling SQLSetConnectAttr with the SQL_ATTR_ASYNC_STMT_EVENT attribute and specifying the event handle.</td>
</tr>
</tbody>
</table>

**Note**

The notification method is not supported with cursor library. An application will receive error message if it attempts to enable cursor library via SQLSetConnectAttr, when the notification method is enabled.

<table>
<thead>
<tr>
<th>SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE (ODBC 3.8)</th>
<th>A SQLUINTEGER value that enables or disables asynchronous execution of selected functions on the connection handle. For more information, see Asynchronous Execution (Polling Method).</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ASYNC_DBC_ENABLE_ON = Enable asynchronous operation for specified connection-related functions.</td>
<td></td>
</tr>
<tr>
<td>SQL_ASYNC_DBC_ENABLE_OFF = (Default) Disable asynchronous operation for specified connection-related functions.</td>
<td></td>
</tr>
<tr>
<td>Setting SQL_ATTR_ASYNC_DBC_FUNCTIONS_ENABLE is always synchronous (that is, it will never return SQL_STILL_EXECUTING).</td>
<td></td>
</tr>
<tr>
<td>Asynchronous execution of statement operations are enabled with SQL_ATTR_ASYNC_ENABLE.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL_ATTR_ASYNC_DBC_PCALLBACK (ODBC 3.8)</th>
<th>A SQLPOINTER value that points to context structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only the Driver Manager can call a driver’s SQLSetStmtAttr function with this attribute.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL_ATTR_ASYNC_DBC_PCONTEXT (ODBC 3.8)</th>
<th>A SQLPOINTER value that points to the context structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only the Driver Manager can call a driver’s</td>
<td></td>
</tr>
</tbody>
</table>
**SQL_ATTR_ASYNC_ENABLE (ODBC 3.0)**

A SQLULEN value that specifies whether a function called with a statement on the specified connection is executed asynchronously:

- **SQL_ASYNC_ENABLE_OFF** = Disable connection level asynchronous execution support for statement operations (the default).
- **SQL_ASYNC_ENABLE_ON** = Enable connection level asynchronous execution support for statement operations.

This attribute can be set whether `SQLGetInfo` with the SQL_ASYNC_MODE information type returns SQL_AM_CONNECTION or SQL_AM_STATEMENT.

---

**SQL_ATTR_AUTO_IPD (ODBC 3.0)**

A read-only SQLUINTEGER value that specifies whether automatic population of the IPD after a call to `SQLPrepare` is supported:

- **SQL_TRUE** = Automatic population of the IPD after a call to `SQLPrepare` is supported by the driver.
- **SQL_FALSE** = Automatic population of the IPD after a call to `SQLPrepare` is not supported by the driver. Servers that do not support prepared statements will not be able to populate the IPD automatically.

If SQL_TRUE is returned for the SQL_ATTR_AUTO_IPD connection attribute, the statement attribute SQL_ATTR_ENABLE_AUTO_IPD can be set to turn automatic population of the IPD on or off. If SQL_ATTR_AUTO_IPD is SQL_FALSE, SQL_ATTR_ENABLE_AUTO_IPD cannot be set to SQL_TRUE. The default value of SQL_ATTR_ENABLE_AUTO_IPD is equal to the value of SQL_ATTR_AUTO_IPD.

This connection attribute can be returned by `SQLGetConnectAttr` but cannot be set by `SQLSetConnectAttr`.

---

**SQL_ATTR_AUTOCOMMIT (ODBC 1.0)**

A SQLUINTEGER value that specifies whether to use autocommit or manual-commit mode:

- **SQL_AUTOCOMMIT_OFF** = The driver uses manual-commit mode, and the application must explicitly commit or roll back transactions with `SQLEndTran`.
- **SQL_AUTOCOMMIT_ON** = The driver uses autocommit mode. Each statement is committed immediately after it is executed. This is the

**SQLSetStmtAttr** function with this attribute.
default. Any open transactions on the connection are committed when SQL_ATTR_AUTOCOMMIT is set to SQL_AUTOCOMMIT_ON to change from manual-commit mode to autocommit mode.

For more information, see Commit Mode.

---

**Important**

Some data sources delete the access plans and close the cursors for all statements on a connection each time a statement is committed; autocommit mode can cause this to happen after each nonquery statement is executed or when the cursor is closed for a query. For more information, see the SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR information types in SQLGetInfo and Effect of Transactions on Cursors and Prepared Statements.

When a batch is executed in autocommit mode, two things are possible. The entire batch can be treated as an autocommitable unit, or each statement in a batch is treated as an autocommitable unit. Certain data sources can support both these behaviors and may provide a way of choosing one or the other. It is driver-defined whether a batch is treated as an autocommitable unit or whether each individual statement within the batch is autocommitable.

---

<table>
<thead>
<tr>
<th>SQL_ATTR_CONNECTION_DEAD (ODBC 3.5)</th>
<th>A read-only SQLINTEGER value that indicates the state of the connection. If SQL_CD_TRUE, the connection has been lost. If SQL_CD_FALSE, the connection is still active.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>SQL_ATTR_CONNECTION_TIMEOUT (ODBC 3.0)</th>
<th>An SQLINTEGER value corresponding to the number of seconds to wait for any request on the connection to complete before returning to the application. The driver should return SQLSTATE HYT00 (Timeout expired) anytime that it is possible to time out in a situation not associated with query execution or login. If ValuePtr is equal to 0 (the default), there is no timeout.</th>
</tr>
</thead>
</table>

| SQL_ATTR_CURRENT_CATALOG (ODBC 2.0) | A character string containing the name of the catalog to be used by the data source. For example, in SQL Server, the catalog is a database, so the driver sends a **USE database** statement to the data source, where **database** is the database specified in *ValuePtr. For a single-tier driver, the |
| SQL_ATTR_DBC_INFO_TOKEN (ODBC 3.8) | A SQLPOINTER value used to set back the connection info token into the DBC handle when SQLRateConnection’s (*pRating) parameter is not equal to 100.

SQL_ATTR_DBC_INFO_TOKEN is set-only. It is not possible to use SQLGetConnectAttr or SQLGetConnectOption to retrieve this value. The Driver Manager’s SQLSetConnectAttr will not accept SQL_ATTR_DBC_INFO_TOKEN, since an application should not set this attribute.

If a driver returns SQL_ERROR after setting SQL_ATTR_DBC_INFO_TOKEN, the connection just obtained from the pool will be freed. The Driver Manager will then try to obtain another connection from the pool. See Developing Connection-Pool Awareness in an ODBC Driver for more information. |
|---|---|
| SQL_ATTR_ENLIST_IN_DTC (ODBC 3.0) | A SQLPOINTER value that specifies whether to use the ODBC driver in distributed transactions coordinated by Microsoft Component Services.

Pass a DTC OLE transaction object that specifies the transaction to export to SQL Server, or SQL_DTC_DONE to end the connection's DTC association.

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_ATTR_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. |
| SQL_ATTR_LOGIN_TIMEOUT (ODBC 1.0) | An SQLINTEGER value corresponding to the number of seconds to wait for a login request to complete before returning to the application. The default is driver-dependent. If ValuePtr is 0, the timeout is disabled and a connection attempt will wait indefinitely. |
If the specified timeout exceeds the maximum login timeout in the data source, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed).

### SQL_ATTR_METADATA_ID (ODBC 3.0)

An SQLINTEGER value that determines how the string arguments of catalog functions are treated.

- **If SQL_TRUE**, the string argument of catalog functions are treated as identifiers. The case is not significant. For non-delimited strings, the driver removes any trailing spaces and the string is folded to uppercase. For delimited strings, the driver removes any leading or trailing spaces and takes literally whatever is between the delimiters.

- If one of these arguments is set to a null pointer, the function returns SQL_ERROR and SQLSTATE HY009 (Invalid use of null pointer).

- **If SQL_FALSE**, the string arguments of catalog functions are not treated as identifiers. The case is significant. They can either contain a string search pattern or not, depending on the argument.

The default value is SQL_FALSE.

The `TableType` argument of `SQLTables`, which takes a list of values, is not affected by this attribute.

SQL_ATTR_METADATA_ID can also be set on the statement level. (It is the only connection attribute that is also a statement attribute.)

For more information, see [Arguments in Catalog Functions](#).

### SQL_ATTR_ODBC_CURSORS (ODBC 2.0)

An SQLULEN value specifying how the Driver Manager uses the ODBC cursor library:

- **SQL_CUR_USE_IF_NEEDED** = The Driver Manager uses the ODBC cursor library only if it is needed. If the driver supports the SQL_FETCH_PRIOR option in `SQLFetchScroll`, the Driver Manager uses the scrolling capabilities of the driver. Otherwise, it uses the ODBC cursor library.

- **SQL_CUR_USE_ODBC** = The Driver Manager uses the ODBC cursor library.

- **SQL_CUR_USE_DRIVER** = The Driver Manager uses the scrolling capabilities of the driver. This is the default setting.

For more information about the ODBC cursor library, see [Appendix F: ODBC Cursor Library](#).
### Warning

The cursor library will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

<table>
<thead>
<tr>
<th>SQL_ATTR_PACKET_SIZE (ODBC 2.0)</th>
<th>An SQLINTEGER value specifying the network packet size in bytes.</th>
</tr>
</thead>
</table>

#### Note

Many data sources either do not support this option or only can return but not set the network packet size.

If the specified size exceeds the maximum packet size or is smaller than the minimum packet size, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed).

If the application sets packet size after a connection has already been made, the driver will return SQLSTATE HY011 (Attribute cannot be set now).

<table>
<thead>
<tr>
<th>SQL_ATTR_QUIET_MODE (ODBC 2.0)</th>
<th>A window handle (HWND).</th>
</tr>
</thead>
</table>

If the window handle is a null pointer, the driver does not display any dialog boxes.

If the window handle is not a null pointer, it should be the parent window handle of the application. This is the default. The driver uses this handle to display dialog boxes.

#### Note

The SQL_ATTR_QUIET_MODE connection attribute does not apply to dialog boxes displayed by `SQLDriverConnect`.

<table>
<thead>
<tr>
<th>SQL_ATTR_TRACE (ODBC 1.0)</th>
<th>An SQLINTEGER value telling the Driver Manager whether to perform tracing:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SQL_OPT_TRACE_OFF = Tracing off (the default)</td>
</tr>
<tr>
<td></td>
<td>SQL_OPT_TRACE_ON = Tracing on</td>
</tr>
</tbody>
</table>

When tracing is on, the Driver Manager writes each ODBC function call to the trace file.
When tracing is on, the Driver Manager can return SQLSTATE IM013 (Trace file error) from any function.

An application specifies a trace file with the SQL_ATTR_TRACEFILE option. If the file already exists, the Driver Manager appends to the file. Otherwise, it creates the file. If tracing is on and no trace file has been specified, the Driver Manager writes to the file SQL.LOG in the root directory.

An application can set the variable ODBCSharedTraceFlag to enable tracing dynamically. Tracing is then enabled for all ODBC applications currently running. If an application turns tracing off, it is turned off only for that application.

If the Trace keyword in the system information is set to 1 when an application calls SQLAllocHandle with a HandleType of SQL_HANDLE_ENV, tracing is enabled for all handles. It is enabled only for the application that called SQLAllocHandle.

Calling SQLSetConnectAttr with an Attribute of SQL_ATTR_TRACE does not require that the ConnectionHandle argument be valid and will not return SQL_ERROR if ConnectionHandle is NULL. This attribute applies to all connections.

SQL_ATTR_TRACEFILE (ODBC 1.0)

A null-terminated character string containing the name of the trace file.

The default value of the SQL_ATTR_TRACEFILE attribute is specified with the TraceFile keyword in the system information. For more information, see ODBC Subkey.

Calling SQLSetConnectAttr with an Attribute of SQL_ATTR_TRACEFILE does not require the ConnectionHandle argument to be valid and will not return SQL_ERROR if ConnectionHandle is invalid. This attribute applies to all connections.

SQL_ATTR_TRANSLATE_LIB (ODBC 1.0)

A null-terminated character string containing the name of a library containing the functions SQLDriverToDataSource and SQLDataSourceToDriver that the driver accesses to perform tasks such as character set translation. This option may be specified only if the driver has connected to the data source. The setting of this attribute will persist across connections. For more information about translating data, see Translation DLLs and Translation DLL Function Reference.
### SQL_ATTR_TRANSLATE_OPTION (ODBC 1.0)

A 32-bit flag value that is passed to the translation DLL. This attribute can be specified only if the driver has connected to the data source. For information about translating data, see Translation DLLs.

---

### SQL_ATTR_TXN_ISOLATION (ODBC 1.0)

A 32-bit bitmask that sets the transaction isolation level for the current connection. An application must call `SQLEndTran` to commit or roll back all open transactions on a connection, before calling `SQLSetConnectAttr` with this option.

The valid values for `ValuePtr` can be determined by calling `SQLGetInfo` with `InfoType` equal to `SQL_TXN_ISOLATION_OPTIONS`.

For a description of transaction isolation levels, see the description of the `SQL_DEFAULT_TXN_ISOLATION` information type in `SQLGetInfo` and Transaction Isolation Levels.

---

[1] These functions can be called asynchronously only if the descriptor is an implementation descriptor, not an application descriptor.

---

**Code Example**

See `SQLConnect`.

---

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocating a handle</td>
<td><code>SQLAllocHandle Function</code></td>
</tr>
<tr>
<td>Returning the setting of a connection attribute</td>
<td><code>SQLGetConnectAttr Function</code></td>
</tr>
</tbody>
</table>

---

**See Also**

- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft
SQLSetConnectOption Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

Summary
In ODBC 3.x, the ODBC 2.0 function SQLSetConnectOption has been replaced by SQLSetConnectAttr. For more information, see SQLSetConnectAttr.

Note
For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions.

Remarks
See ODBC 64-Bit Information, if your application will run on a 64-bit operating system.

Note
The attribute SQL_ASYNC_DBC_FUNCTION_ENABLE introduced in ODBC 3.8 is not supported by SQLSetConnectOption. Applications that use the asynchronous operation on connection handle must use SQLSetConnectAttr.

See Also
ODBC API Reference
ODBC Header Files

SQLSetCursorName Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLSetCursorName associates a cursor name with an active statement. If an application does not call SQLSetCursorName, the driver generates cursor names as needed for SQL statement processing.

Syntax
Arguments

StatementHandle
[Input] Statement handle.

CursorName
[Input] Cursor name. For efficient processing, the cursor name should not include any leading or trailing spaces in the cursor name, and if the cursor name includes a delimited identifier, the delimiter should be positioned as the first character in the cursor name.

NameLength
[Input] Length in characters of *CursorName.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLSetCursorName returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLSetCursorName and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right truncated</td>
<td>The cursor name exceeded the maximum limit, so only the maximum allowable number of characters was used.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>The statement corresponding to StatementHandle was already in an executed or cursor-positioned state.</td>
</tr>
<tr>
<td>34000</td>
<td>Invalid cursor name</td>
<td>The cursor name specified in *CursorName was invalid because it exceeded the maximum length as defined by</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Message</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3C000</td>
<td>Duplicate cursor name</td>
<td>The cursor name specified in *CursorName already exists.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>(DM) The argument CursorName was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This synchronous function was still executing when the SQLSetCursorName function was called. (DM) An asynchronously executing function was called for the StatementHandle and was still executing when this function was called. (DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The argument NameLength was less than 0 but not equal to SQL_NTS.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
</tbody>
</table>
Comments

Cursor names are used only in positioned update and delete statements (for example, `UPDATE table-name ... WHERE CURRENT OF cursor-name`). For more information, see Positioned Update and Delete Statements. If the application does not call `SQLSetCursorName` to define a cursor name, on execution of a query statement the driver generates a name that begins with the letters SQL_CUR and does not exceed 18 characters in length.

All cursor names within the connection must be unique. The maximum length of a cursor name is defined by the driver. For maximum interoperability, it is recommended that applications limit cursor names to no more than 18 characters. In ODBC 3.x, if a cursor name is a quoted identifier, it is treated in a case-sensitive manner and it can contain characters that the syntax of SQL would not permit or would treat specially, such as blanks or reserved keywords. If a cursor name must be treated in a case-sensitive manner, it must be passed as a quoted identifier.

A cursor name that is set either explicitly or implicitly remains set until the statement with which it is associated is dropped, using `SQLFreeHandle`. `SQLSetCursorName` can be called to rename a cursor on a statement as long as the cursor is in an allocated or prepared state.

Code Example

In the following example, an application uses `SQLSetCursorName` to set a cursor name for a statement. It then uses that statement to retrieve results from the CUSTOMERS table. Finally, it performs a positioned update to change the phone number of John Smith. Note that the application uses different statement handles for the `SELECT` and `UPDATE` statements.

For another code example, see `SQLSetPos`.

```c
#define NAME_LEN 50
#define PHONE_LEN 10

SQLHSTMT hstmtSelect,
SQLHSTMT hstmtUpdate;
SQLRETURN retcode;
SQLHDBC hdbc;
SQLCHAR szName[NAME_LEN], szPhone[PHONE_LEN];
SQLINTEGER cbName, cbPhone;

/* Allocate the statements and set the cursor name. */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmtSelect);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmtUpdate);
SQLSetCursorName(hstmtSelect, "C1", SQL_NTS);

/* SELECT the result set and bind its columns to local buffers. */
SQLExecDirect(hstmtSelect,
    "SELECT NAME, PHONE FROM CUSTOMERS",
    SQL_NTS);
SQLBindCol(hstmtSelect, 1, SQL_C_CHAR, szName, NAME_LEN, &cbName);
SQLBindCol(hstmtSelect, 2, SQL_C_CHAR, szPhone, PHONE_LEN, &cbPhone);
```
/* Read through the result set until the cursor is */
/* positioned on the row for John Smith. */

do
  retcode = SQLFetch(hstmtSelect);
while ((retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) &&
  (strcmp(szName, "Smith, John") != 0));

/* Perform a positioned update of John Smith's name. */
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
  SQLExecDirect(hstmtUpdate,
    "UPDATE EMPLOYEE SET PHONE="2064890154" WHERE CURRENT OF C1",
    SQL_NTS);
}

## Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executing an SQL statement</td>
<td>SQLExecDirect Function</td>
</tr>
<tr>
<td>Executing a prepared SQL statement</td>
<td>SQLExecute Function</td>
</tr>
<tr>
<td>Returning a cursor name</td>
<td>SQLGetCursorName Function</td>
</tr>
<tr>
<td>Setting cursor scrolling options</td>
<td>SQLSetScrollOptions Function</td>
</tr>
</tbody>
</table>

## See Also

- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft

## SQLSetDescField Function

### Conformance

Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

### Summary

SQLSetDescField sets the value of a single field of a descriptor record.
Syntax

```c
SQLRETURN SQLSetDescField(
    SQLHDESC  DescriptorHandle,
    SQLSMALLINT  RecNumber,
    SQLSMALLINT  FieldIdentifier,
    SQLPOINTER  ValuePtr,
    SQLINTEGER  BufferLength);
```

Arguments

**DescriptorHandle**

[Input] Descriptor handle.

**RecNumber**

[Input] Indicates the descriptor record containing the field that the application seeks to set. Descriptor records are numbered from 0, with record number 0 being the bookmark record. The `RecNumber` argument is ignored for header fields.

**FieldIdentifier**

[Input] Indicates the field of the descriptor whose value is to be set. For more information, see "FieldIdentifier Argument" in the "Comments" section.

**ValuePtr**

[Input] Pointer to a buffer containing the descriptor information, or an integer value. The data type depends on the value of `FieldIdentifier`. If `ValuePtr` is an integer value, it may be considered as 8 bytes (SQLLEN), 4 bytes (SQLINTEGER) or 2 bytes (SQLSMALLINT), depending on the value of the `FieldIdentifier` argument.

**BufferLength**

[Input] If `FieldIdentifier` is an ODBC-defined field and `ValuePtr` points to a character string or a binary buffer, this argument should be the length of `ValuePtr`. For character string data, this argument should contain the number of bytes in the string.

If `FieldIdentifier` is an ODBC-defined field and `ValuePtr` is an integer, `BufferLength` is ignored.

If `FieldIdentifier` is a driver-defined field, the application indicates the nature of the field to the Driver Manager by setting the `BufferLength` argument. `BufferLength` can have the following values:

- If `ValuePtr` is a pointer to a character string, then `BufferLength` is the length of the string or SQL_NTS.
- If `ValuePtr` is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in `BufferLength`. This places a negative value in `BufferLength`.
- If `ValuePtr` is a pointer to a value other than a character string or a binary string, then `BufferLength` should have the value SQL_IS_POINTER.
- If `ValuePtr` contains a fixed-length value, then `BufferLength` is either SQL_IS_INTEGER, SQL_IS_UINTEGER, SQL_IS_SMALLINT, or SQL_IS_USMALLINT, as appropriate.
Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When **SQLSetDescField** returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling **SQLGetDiagRec** with a **HandleType** of SQL_HANDLE_DESC and a **Handle of DescriptorHandle**. The following table lists the SQLSTATE values commonly returned by **SQLSetDescField** and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed</td>
<td>The driver did not support the value specified in *ValuePtr (if ValuePtr was a pointer) or the value in ValuePtr (if ValuePtr was an integer value), or *ValuePtr was invalid because of implementation working conditions, so the driver substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>The FieldIdentifier argument was a record field, the RecNumber argument was 0, and the DescriptorHandle argument referred to an IPD handle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The RecNumber argument was less than 0, and the DescriptorHandle argument referred to an ARD or an APD.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The RecNumber argument was greater than the maximum number of columns or parameters that the data source can support, and the DescriptorHandle argument referred to an APD or ARD.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DM) The FieldIdentifier argument was SQL_DESC_COUNT, and *ValuePtr argument was less than 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The RecNumber argument was equal to 0, and the DescriptorHandle argument referred to an implicitly allocated APD. (This error does not occur with an explicitly allocated application descriptor, because it is not known whether an explicitly allocated application descriptor is an APD or ARD until execute time.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link</td>
<td>The communication link between the driver and the data source is unhealthy.</td>
</tr>
<tr>
<td>Failure Source</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>22001</td>
<td>String data, right truncated. The <code>FieldIdentifier</code> argument was SQL_DESC_NAME, and the <code>BufferLength</code> argument was a value larger than SQL_MAX_IDENTIFIER_LEN.</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>General error. An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <em>MessageText</em> buffer describes the error and its cause.</td>
<td></td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error. The driver was unable to allocate memory required to support execution or completion of the function.</td>
<td></td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error. (DM) The <code>DescriptorHandle</code> was associated with a <code>StatementHandle</code> for which an asynchronously executing function (not this one) was called and was still executing when this function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for the <code>StatementHandle</code> with which the <code>DescriptorHandle</code> was associated and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) An asynchronously executing function was called for the connection handle that is associated with the <code>DescriptorHandle</code>. This asynchronous function was still executing when the <code>SQLSetDescField</code> function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for one of the statement handles associated with the <code>DescriptorHandle</code> and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.</td>
<td></td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error. The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
<td></td>
</tr>
<tr>
<td>HY016</td>
<td>Cannot modify an implementation row descriptor. The <code>DescriptorHandle</code> argument was associated with an IRD, and the <code>FieldIdentifier</code> argument was not SQL_DESC_ARRAY_STATUS_PTR or SQL_DESC_ROWS_PROCESSED_PTR.</td>
<td></td>
</tr>
<tr>
<td>HY021</td>
<td>Inconsistent descriptor information. The <code>SQL_DESC_TYPE</code> and <code>SQL_DESC_DATETIME_INTERVAL_CODE</code> fields do not form a valid ODBC SQL type or a valid driver-specific SQL type (for IPDs) or a valid ODBC C type (for APDs or ARDs). Descriptor information checked during a consistency check was not consistent. (See &quot;Consistency Check&quot; in <code>SQLSetDescRec</code>.)</td>
<td></td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) *ValuePtr is a character string, and BufferLength was less than zero but was not equal to SQL_NTS. (DM) The driver was an ODBC 2.x driver, the descriptor was an ARD, the ColumnNumber argument was set to 0, and the value specified for the argument BufferLength was not equal to 4.</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------</td>
<td>---</td>
</tr>
<tr>
<td>HY091</td>
<td>Invalid descriptor field identifier</td>
<td>The value specified for the FieldIdentifier argument was not an ODBC-defined field and was not an implementation-defined value. The FieldIdentifier argument was invalid for the DescriptorHandle argument. The FieldIdentifier argument was a read-only, ODBC-defined field.</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute/option identifier</td>
<td>The value in *ValuePtr was not valid for the FieldIdentifier argument. The FieldIdentifier argument was SQL_DESC_UNNAMED, and ValuePtr was SQL_NAMED.</td>
</tr>
<tr>
<td>HY105</td>
<td>Invalid parameter type</td>
<td>(DM) The value specified for the SQL_DESC_PARAMETER_TYPE field was invalid. (For more information, see the &quot;InputOutputType Argument&quot; section in SQLBindParameter.)</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see What's New in ODBC 3.8.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the DescriptorHandle does not support the function.</td>
</tr>
</tbody>
</table>

**Comments**

An application can call SQLSetDescField to set any descriptor field one at a time. One call to SQLSetDescField sets a single field in a single descriptor. This function can be called to set any field in any descriptor type, provided the field can be set. (See the table later in this section.)
If a call to `SQLSetDescField` fails, the contents of the descriptor record identified by the `RecNumber` argument are undefined.

Other functions can be called to set multiple descriptor fields with a single call of the function. The `SQLSetDescRec` function sets a variety of fields that affect the data type and buffer bound to a column or parameter (the `SQL_DESC_TYPE`, `SQL_DESC_DATETIME_INTERVAL_CODE`, `SQL_DESC_OCTET_LENGTH`, `SQL_DESC_PRECISION`, `SQL_DESC_SCALE`, `SQL_DESC_DATA_PTR`, `SQL_DESC_OCTET_LENGTH_PTR`, and `SQL_DESC_INDICATOR_PTR` fields). `SQLBindCol` or `SQLBindParameter` can be used to make a complete specification for the binding of a column or parameter. These functions set a specific group of descriptor fields with one function call.

`SQLSetDescField` can be called to change the binding buffers by adding an offset to the binding pointers (`SQL_DESC_DATA_PTR`, `SQL_DESC_INDICATOR_PTR`, or `SQL_DESC_OCTET_LENGTH_PTR`). This changes the binding buffers without calling `SQLBindCol` or `SQLBindParameter`, which allows an application to change `SQL_DESC_DATA_PTR` without changing other fields, such as `SQL_DESC_DATA_TYPE`.

If an application calls `SQLSetDescField` to set any field other than `SQL_DESC_COUNT` or the deferred fields `SQL_DESC_DATA_PTR`, `SQL_DESC_OCTET_LENGTH_PTR`, or `SQL_DESC_INDICATOR_PTR`, the record becomes unbound.

Descriptor header fields are set by calling `SQLSetDescField` with the appropriate `FieldIdentifier`. Many header fields are also statement attributes, so they can also be set by a call to `SQLSetStmtAttr`. This allows applications to set a descriptor field without first obtaining a descriptor handle. When `SQLSetDescField` is called to set a header field, the `RecNumber` argument is ignored.

A `RecNumber` of 0 is used to set bookmark fields.

The statement attribute `SQL_ATTR_USE_BOOKMARKS` should always be set before calling `SQLSetDescField` to set bookmark fields. While this is not mandatory, it is strongly recommended.

**Sequence of Setting Descriptor Fields**

When setting descriptor fields by calling `SQLSetDescField`, the application must follow a specific sequence:

1. The application must first set the `SQL_DESC_TYPE`, `SQL_DESC_CONCISE_TYPE`, or `SQL_DESC_DATETIME_INTERVAL_CODE` field.

2. After one of these fields has been set, the application can set an attribute of a data type, and the driver sets data type attribute fields to the appropriate default values for the data type. Automatic defaulting of type attribute fields ensures that the descriptor is always ready to use once the application has specified a data type. If the application explicitly sets a data type attribute, it is overriding the default attribute.

3. After one of the fields listed in step 1 has been set, and data type attributes have been set, the application can set `SQL_DESC_DATA_PTR`. This prompts a consistency check of descriptor fields. If the application changes the data type or attributes after setting the `SQL_DESC_DATA_PTR` field, the driver sets `SQL_DESC_DATA_PTR` to a null pointer, unbinding the record. This forces the application to complete the proper steps in sequence, before the descriptor record is usable.
Initialization of Descriptor Fields

When a descriptor is allocated, the fields in the descriptor can be initialized to a default value, be initialized without a default value, or be undefined for the type of descriptor. The following tables indicate the initialization of each field for each type of descriptor, with "D" indicating that the field is initialized with a default, and "ND" indicating that the field is initialized without a default. If a number is shown, the default value of the field is that number. The tables also indicate whether a field is read/write (R/W) or read-only (R).

The fields of an IRD have a default value only after the statement has been prepared or executed and the IRD has been populated, not when the statement handle or descriptor has been allocated. Until the IRD has been populated, any attempt to gain access to a field of an IRD will return an error.

Some descriptor fields are defined for one or more, but not all, of the descriptor types (ARDs and IRDs, and APDs and IPDs). When a field is undefined for a type of descriptor, it is not needed by any of the functions that use that descriptor.

The fields that can be accessed by `SQLGetDescField` cannot necessarily be set by `SQLSetDescField`. Fields that can be set by `SQLSetDescField` are listed in the following tables.

The initialization of header fields is outlined in the table that follows.

<table>
<thead>
<tr>
<th>Header field name</th>
<th>Type</th>
<th>R/W</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_ALLOC_TYPE</td>
<td>SQLSMALLINT</td>
<td>ARD: R APD: R IRD: R IPD: R</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ARD: SQL_DESC_ALLOC_AUTO for implicit or SQL_DESC_ALLOC_USER for explicit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>APD: SQL_DESC_ALLOC_AUTO for implicit or SQL_DESC_ALLOC_USER for explicit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IRD: SQL_DESC_ALLOC_AUTO</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IPD: SQL_DESC_ALLOC_AUTO</td>
<td></td>
</tr>
<tr>
<td>SQL_DESC_ARRAY_SIZE</td>
<td>SQLULEN</td>
<td>ARD: R/W APD: R/W IRD: Unused IPD: Unused</td>
<td></td>
</tr>
<tr>
<td>SQL_DESC_ARRAY_STATUS_PTR</td>
<td>SQLUSMALLINT*</td>
<td>ARD: R/W APD: R/W IRD: R/W IPD: R/W</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ARD: Null ptr APD: Null ptr IRD: Null ptr IPD: Null ptr</td>
<td></td>
</tr>
</tbody>
</table>
These fields are defined only when the IPD is automatically populated by the driver. If not, they are undefined. If an application attempts to set these fields, SQLSTATE HY091 (Invalid descriptor field identifier) will be returned.

The initialization of record fields is as shown in the following table.

<table>
<thead>
<tr>
<th>Record field name</th>
<th>Type</th>
<th>R/W</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Type</td>
<td>ARD</td>
<td>APD</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>--------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SQL_DESC_ROWVER</td>
<td>SQLSMALLINT</td>
<td>ARD: Unused</td>
<td>ARD: Unused</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>FieldIdentifier Argument</td>
<td>SQL_DESC_UNSIGNED</td>
<td>SQL_DESC_UPDATABLE</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------</td>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td>SQLSMALLINT</td>
<td>SQLSMALLINT</td>
<td>SQLSMALLINT</td>
<td></td>
</tr>
<tr>
<td>ARD: Unused</td>
<td>ARD: Unused</td>
<td>ARD: Unused</td>
<td></td>
</tr>
<tr>
<td>APD: Unused</td>
<td>APD: Unused</td>
<td>APD: Unused</td>
<td></td>
</tr>
<tr>
<td>IRD: R/W</td>
<td>IRD: R</td>
<td>IRD: D</td>
<td></td>
</tr>
<tr>
<td>IPD: R/W</td>
<td>IPD: R</td>
<td>IPD: ND</td>
<td></td>
</tr>
<tr>
<td>ARD: Unused</td>
<td>ARD: Unused</td>
<td>ARD: Unused</td>
<td></td>
</tr>
<tr>
<td>APD: Unused</td>
<td>APD: Unused</td>
<td>APD: Unused</td>
<td></td>
</tr>
<tr>
<td>IRD: R</td>
<td>IRD: R</td>
<td>IRD: D[1]</td>
<td></td>
</tr>
<tr>
<td>IPD: R</td>
<td>IPD: R</td>
<td>IPD: ND</td>
<td></td>
</tr>
<tr>
<td>ARD: Unused</td>
<td>ARD: Unused</td>
<td>ARD: Unused</td>
<td></td>
</tr>
<tr>
<td>APD: Unused</td>
<td>APD: Unused</td>
<td>APD: Unused</td>
<td></td>
</tr>
<tr>
<td>IRD: R</td>
<td>IRD: R</td>
<td>IRD: D[1]</td>
<td></td>
</tr>
<tr>
<td>IPD: R</td>
<td>IPD: R</td>
<td>IPD: ND</td>
<td></td>
</tr>
</tbody>
</table>

[1] These fields are defined only when the IPD is automatically populated by the driver. If not, they are undefined. If an application attempts to set these fields, SQLSTATE HY091 (Invalid descriptor field identifier) will be returned.

[2] The SQL_DESC_DATA_PTR field in the IPD can be set to force a consistency check. In a subsequent call to SQLGetDescField or SQLGetDescRec, the driver is not required to return the value that SQL_DESC_DATA_PTR was set to.

FieldIdentifier Argument

The FieldIdentifier argument indicates the descriptor field to be set. A descriptor contains the descriptor header, consisting of the header fields described in the next section, "Header Fields," and zero or more descriptor records, consisting of the record fields described in the section following the "Header Fields" section.

Header Fields

Each descriptor has a header consisting of the following fields:

**SQL_DESC_ALLOC_TYPE [All]**
This read-only SQLSMALLINT header field specifies whether the descriptor was allocated automatically by the driver or explicitly by the application. The application can obtain, but not modify, this field. The field is set to SQL_DESC_ALLOC_AUTO by the driver if the descriptor was automatically allocated by the driver. It is set to SQL_DESC_ALLOC_USER by the driver if the descriptor was explicitly allocated by the application.

**SQL_DESC_ARRAY_SIZE [Application descriptors]**
In ARDs, this SQUALEN header field specifies the number of rows in the rowset. This is the number of rows to be returned by a call to SQLFetch or SQLFetchScroll or to be operated on
by a call to SQLBulkOperations or SQLSetPos.

In APDs, this SQLULEN header field specifies the number of values for each parameter.

The default value of this field is 1. If SQL_DESC_ARRAY_SIZE is greater than 1, 
SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR of 
the APD or ARD point to arrays. The cardinality of each array is equal to the value of this field.

This field in the ARD can also be set by calling SQLSetStmtAttr with the 
SQL_ATTR_ROW_ARRAY_SIZE attribute. This field in the APD can also be set by calling 
SQLSetStmtAttr with the SQL_ATTR_PARAMSET_SIZE attribute.

**SQL_DESC_ARRAY_STATUS_PTR [All]**

For each descriptor type, this SQLUSMALLINT * header field points to an array of 
SQLUSMALLINT values. These arrays are named as follows: row status array (IRD), parameter 
status array (IPD), row operation array (ARD), and parameter operation array (APD).

In the IRD, this header field points to a row status array containing status values after a call to 
SQLBulkOperations, SQLFetch, SQLFetchScroll, or SQLSetPos. The array has as many 
elements as there are rows in the rowset. The application must allocate an array of 
SQLUSMALLINTs and set this field to point to the array. The field is set to a null pointer by 
default. The driver will populate the array — unless the SQL_DESC_ARRAY_STATUS_PTR field is 
set to a null pointer, in which case no status values are generated and the array is not 
populated.

⚠️ **Caution**

Driver behavior is undefined if the application sets the elements of the row status array 
pointed to by the SQL_DESC_ARRAY_STATUS_PTR field of the IRD.

The array is initially populated by a call to SQLBulkOperations, SQLFetch, SQLFetchScroll, 
or SQLSetPos. If the call did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the 
contents of the array pointed to by this field are undefined. The elements in the array can 
contain the following values:

- **SQL_ROW_SUCCESS**: The row was successfully fetched and has not changed since it was 
  last fetched.
- **SQL_ROW_SUCCESS_WITH_INFO**: The row was successfully fetched and has not changed 
  since it was last fetched. However, a warning was returned about the row.
- **SQL_ROW_ERROR**: An error occurred while fetching the row.
- **SQL_ROW_UPDATED**: The row was successfully fetched and has been updated since it was 
  last fetched. If the row is fetched again, its status is SQL_ROW_SUCCESS.
- **SQL_ROW_DELETED**: The row has been deleted since it was last fetched.
- **SQL_ROW_ADDED**: The row was inserted by SQLBulkOperations. If the row is fetched 
  again, its status is SQL_ROW_SUCCESS.
- **SQL_ROW_NOROW**: The rowset overlapped the end of the result set, and no row was 
  returned that corresponded to this element of the row status array.

This field in the IRD can also be set by calling SQLSetStmtAttr with the 
SQL_ATTR_ROW_STATUS_PTR attribute.
The SQL_DESC_ARRAY_STATUS_PTR field of the IRD is valid only after SQL_SUCCESS or SQL_SUCCESS_WITH_INFO has been returned. If the return code is not one of these, the location pointed to by SQL_DESC_ROWS_PROCESSED_PTR is undefined.

In the IPD, this header field points to a parameter status array containing status information for each set of parameter values after a call to SQLExecute or SQLExecDirect. If the call to SQLExecute or SQLExecDirect did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the array pointed to by this field are undefined. The application must allocate an array of SQLUSMALLINTs and set this field to point to the array. The driver will populate the array — unless the SQL_DESC_ARRAY_STATUS_PTR field is set to a null pointer, in which case no status values are generated and the array is not populated. The elements in the array can contain the following values:

- SQL_PARAM_SUCCESS: The SQL statement was successfully executed for this set of parameters.
- SQL_PARAM_SUCCESS_WITH_INFO: The SQL statement was successfully executed for this set of parameters; however, warning information is available in the diagnostics data structure.
- SQL_PARAM_ERROR: An error occurred in processing this set of parameters. Additional error information is available in the diagnostics data structure.
- SQL_PARAM_UNUSED: This parameter set was unused, possibly due to the fact that some previous parameter set caused an error that aborted further processing, or because SQL_PARAM_IGNORE was set for that set of parameters in the array specified by the SQL_DESC_ARRAY_STATUS_PTR field of the APD.
- SQL_PARAM_DIAG_UNAVAILABLE: Diagnostic information is not available. An example of this is when the driver treats arrays of parameters as a monolithic unit and so does not generate this level of error information.

This field in the IPD can also be set by calling SQLSetStmtAttr with the SQL_ATTR_PARAM_STATUS_PTR attribute.

In the ARD, this header field points to a row operation array of values that can be set by the application to indicate whether this row is to be ignored for SQLSetPos operations. The elements in the array can contain the following values:

- SQL_ROW_PROCEED: The row is included in the bulk operation using SQLSetPos. (This setting does not guarantee that the operation will occur on the row. If the row has the status SQL_ROW_ERROR in the IRD row status array, the driver might not be able to perform the operation in the row.)
- SQL_ROW_IGNORE: The row is excluded from the bulk operation using SQLSetPos.

If no elements of the array are set, all rows are included in the bulk operation. If the value in the SQL_DESC_ARRAY_STATUS_PTR field of the ARD is a null pointer, all rows are included in the bulk operation; the interpretation is the same as if the pointer pointed to a valid array and all elements of the array were SQL_ROW_PROCEED. If an element in the array is set to SQL_ROW_IGNORE, the value in the row status array for the ignored row is not changed.

This field in the ARD can also be set by calling SQLSetStmtAttr with the SQL_ATTR_ROW_OPERATION_PTR attribute.
In the APD, this header field points to a parameter operation array of values that can be set by the application to indicate whether this set of parameters is to be ignored when `SQLExecute` or `SQLExecDirect` is called. The elements in the array can contain the following values:

- **SQL_PARAM_PROCEED**: The set of parameters is included in the `SQLExecute` or `SQLExecDirect` call.
- **SQL_PARAM_IGNORE**: The set of parameters is excluded from the `SQLExecute` or `SQLExecDirect` call.

If no elements of the array are set, all sets of parameters in the array are used in the `SQLExecute` or `SQLExecDirect` calls. If the value in the `SQL_DESC_ARRAY_STATUS_PTR` field of the APD is a null pointer, all sets of parameters are used; the interpretation is the same as if the pointer pointed to a valid array and all elements of the array were `SQL_PARAM_PROCEED`.

This field in the APD can also be set by calling `SQLSetStmtAttr` with the `SQL_ATTR_PARAM_OPERATION_PTR` attribute.

**SQL_DESC_BIND_OFFSET_PTR [Application descriptors]**

This `SQLLEN *` header field points to the binding offset. It is set to a null pointer by default. If this field is not a null pointer, the driver dereferences the pointer and adds the dereferenced value to each of the deferred fields that has a non-null value in the descriptor record (`SQL_DESC_DATA_PTR`, `SQL_DESC_INDICATOR_PTR`, and `SQL_DESC_OCTET_LENGTH_PTR`) at fetch time and uses the new pointer values when binding.

The binding offset is always added directly to the values in the `SQL_DESC_DATA_PTR`, `SQL_DESC_INDICATOR_PTR`, and `SQL_DESC_OCTET_LENGTH_PTR` fields. If the offset is changed to a different value, the new value is still added directly to the value in each descriptor field. The new offset is not added to the field value plus any earlier offset.

This field is a **deferred field**: It is not used at the time it is set but is used at a later time by the driver when it needs to determine addresses for data buffers.

This field in the ARD can also be set by calling `SQLSetStmtAttr` with the `SQL_ATTR_ROW_BIND_OFFSET_PTR` attribute. This field in the ARD can also be set by calling `SQLSetStmtAttr` with the `SQL_ATTR_PARAM_BIND_OFFSET_PTR` attribute.

For more information, see the description of row-wise binding in `SQLFetchScroll` and `SQLBindParameter`.

**SQL_DESC_BIND_TYPE [Application descriptors]**

This `SQLINTEGER` header field sets the binding orientation to be used for binding either columns or parameters.

In ARDs, this field specifies the binding orientation when `SQLFetchScroll` or `SQLFetch` is called on the associated statement handle.

To select column-wise binding for columns, this field is set to `SQL_BIND_BY_COLUMN` (the default).

This field in the ARD can also be set by calling `SQLSetStmtAttr` with the `SQL_ATTR_ROW_BIND_TYPE Attribute`.

In APDs, this field specifies the binding orientation to be used for dynamic parameters.

To select column-wise binding for parameters, this field is set to `SQL_BIND_BY_COLUMN` (the default).
This field in the APD can also be set by calling `SQLSetStmtAttr` with the `SQL_ATTR_PARAM_BIND_TYPE` attribute.

**SQL_DESC_COUNT [All]**

This SQLSMALLINT header field specifies the 1-based index of the highest-numbered record that contains data. When the driver sets the data structure for the descriptor, it must also set the SQL_DESC_COUNT field to show how many records are significant. When an application allocates an instance of this data structure, it does not have to specify how many records to reserve room for. As the application specifies the contents of the records, the driver takes any required action to ensure that the descriptor handle refers to a data structure of the adequate size.

SQL_DESC_COUNT is not a count of all data columns that are bound (if the field is in an ARD) or of all parameters that are bound (if the field is in an APD), but the number of the highest-numbered record. If the highest-numbered column or parameter is unbound, then SQL_DESC_COUNT is changed to the number of the next highest-numbered column or parameter. If a column or a parameter with a number that is less than the number of the highest-numbered column is unbound (by calling `SQLBindCol` with the `TargetValuePtr` argument set to a null pointer, or `SQLBindParameter` with the `ParameterValuePtr` argument set to a null pointer), SQL_DESC_COUNT is not changed. If additional columns or parameters are bound with numbers greater than the highest-numbered record that contains data, the driver automatically increases the value in the SQL_DESC_COUNT field. If all columns are unbound by calling `SQLFreeStmt` with the SQL_UNBIND option, the SQL_DESC_COUNT fields in the ARD and IRD are set to 0. If `SQLFreeStmt` is called with the SQL_RESET_PARAMS option, the SQL_DESC_COUNT fields in the APD and IPD are set to 0.

The value in SQL_DESC_COUNT can be set explicitly by an application by calling `SQLSetDescField`. If the value in SQL_DESC_COUNT is explicitly decreased, all records with numbers greater than the new value in SQL_DESC_COUNT are effectively removed. If the value in SQL_DESC_COUNT is explicitly set to 0 and the field is in an ARD, all data buffers except a bound bookmark column are released.

The record count in this field of an ARD does not include a bound bookmark column. The only way to unbind a bookmark column is to set the SQL_DESC_DATA_PTR field to a null pointer.

**SQL_DESC_ROWS_PROCESSED_PTR [Implementation descriptors]**

In an IRD, this SQLULEN * header field points to a buffer containing the number of rows fetched after a call to `SQLFetch` or `SQLFetchScroll`, or the number of rows affected in a bulk operation performed by a call to `SQLBulkOperations` or `SQLSetPos`, including error rows.

In an IPD, this SQUINTTEGRER * header field points to a buffer containing the number of sets of parameters that have been processed, including error sets. No number will be returned if this is a null pointer.

SQL_DESC_ROWS_PROCESSED_PTR is valid only after SQL_SUCCESS or SQL_SUCCESS_WITH_INFO has been returned after a call to `SQLFetch` or `SQLFetchScroll` (for an IRD field) or `SQLExecute`, `SQLExecDirect`, or `SQLParamData` (for an IPD field). If the call that fills in the buffer pointed to by this field does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined, unless it returns SQL_NO_DATA, in which case the value in the buffer is set to 0.

This field in the ARD can also be set by calling `SQLSetStmtAttr` with the `SQL_ATTR_ROWS_Fetched_PTR` attribute. This field in the APD can also be set by calling `SQLSetStmtAttr` with the `SQL_ATTR_PARAMS_PROCESSED_PTR` attribute.

The buffer pointed to by this field is allocated by the application. It is a deferred output buffer that is set by the driver. It is set to a null pointer by default.
Record Fields

Each descriptor contains one or more records consisting of fields that define either column data or dynamic parameters, depending on the type of descriptor. Each record is a complete definition of a single column or parameter.

**SQL_DESC_AUTO_UNIQUE_VALUE [IRDs]**
This read-only SQLINTEGER record field contains SQL_TRUE if the column is an auto-incrementing column, or SQL_FALSE if the column is not an auto-incrementing column. This field is read-only, but the underlying auto-incrementing column is not necessarily read-only.

**SQL_DESC_BASE_COLUMN_NAME [IRDs]**
This read-only SQLCHAR * record field contains the base column name for the result set column. If a base column name does not exist (as in the case of columns that are expressions), this variable contains an empty string.

**SQL_DESC_BASE_TABLE_NAME [IRDs]**
This read-only SQLCHAR * record field contains the base table name for the result set column. If a base table name cannot be defined or is not applicable, this variable contains an empty string.

**SQL_DESC_CASE_SENSITIVE [Implementation descriptors]**
This read-only SQLINTEGER record field contains SQL_TRUE if the column or parameter is treated as case-sensitive for collations and comparisons, or SQL_FALSE if the column is not treated as case-sensitive for collations and comparisons or if it is a noncharacter column.

**SQL_DESC_CATALOG_NAME [IRDs]**
This read-only SQLCHAR * record field contains the catalog for the base table that contains the column. The return value is driver-dependent if the column is an expression or if the column is part of a view. If the data source does not support catalogs or the catalog cannot be determined, this variable contains an empty string.

**SQL_DESC_CONCISE_TYPE [All]**
This SQLSMALLINT header field specifies the concise data type for all data types, including the datetime and interval data types.

The values in the SQL_DESC_CONCISE_TYPE, SQL_DESC_TYPE, and SQL_DESC_DATETIME_INTERVAL_CODE fields are interdependent. Each time one of the fields is set, the other must also be set. SQL_DESC_CONCISE_TYPE can be set by a call to SQLBindCol or SQLBindParameter, or SQLSetDescField. SQL_DESC_TYPE can be set by a call to SQLSetDescField or SQLSetDescRec.

If SQL_DESC_CONCISE_TYPE is set to a concise data type other than an interval or datetime data type, the SQL_DESC_TYPE field is set to the same value and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to 0.

If SQL_DESC_CONCISE_TYPE is set to the concise datetime or interval data type, the SQL_DESC_TYPE field is set to the corresponding verbose type (SQL_DATETIME or SQL_INTERVAL) and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to the appropriate subcode.

**SQL_DESC_DATA_PTR [Application descriptors and IPDs]**
This SQLPOINTER record field points to a variable that will contain the parameter value (for APDs) or the column value (for ARDs). This field is a deferred field. It is not used at the time it is set but is used at a later time by the driver to retrieve data.
The column specified by the SQL_DESC_DATA_PTR field of the ARD is unbound if the TargetValuePtr argument in a call to SQLBindCol is a null pointer or if the SQL_DESC_DATA_PTR field in the ARD is set by a call to SQLSetDescField or SQLSetDescRec to a null pointer. Other fields are not affected if the SQL_DESC_DATA_PTR field is set to a null pointer.

If the call to SQLFetch or SQLFetchScroll that fills in the buffer pointed to by this field did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

Whenever the SQL_DESC_DATA_PTR field of an APD, ARD, or IPD is set, the driver checks that the value in the SQL_DESC_TYPE field contains one of the valid ODBC C data types or a driver-specific data type, and that all other fields affecting the data types are consistent. Prompting a consistency check is the only use of the SQL_DESC_DATA_PTR field of an IPD. Specifically, if an application sets the SQL_DESC_DATA_PTR field of an IPD and later calls SQLGetDescField on this field, it is not necessarily returned the value that it had set. For more information, see "Consistency Checks" in SQLSetDescRec.

SQL_DESC_DATETIME_INTERVAL_CODE [All]
This SQLSMALLINT record field contains the subcode for the specific datetime or interval data type when the SQL_DESC_TYPE field is SQL_DATETIME or SQL_INTERVAL. This is true for both SQL and C data types. The code consists of the data type name with "CODE" substituted for either "TYPE" or "C_TYPE" (for datetime types), or "CODE" substituted for "INTERVAL" or "C_INTERVAL" (for interval types).

If SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE in an application descriptor are set to SQL_C_DEFAULT and the descriptor is not associated with a statement handle, the contents of SQL_DESC_DATETIME_INTERVAL_CODE are undefined.

This field can be set for the datetime data types listed in the following table.

<table>
<thead>
<tr>
<th>Datetime types</th>
<th>DATETIME_INTERVAL_CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_TYPE_DATE/SQL_C_TYPE_DATE</td>
<td>SQL_CODE_DATE</td>
</tr>
<tr>
<td>SQL_TYPE_TIME/SQL_C_TYPE_TIME</td>
<td>SQL_CODE_TIME</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP/SQL_C_TYPE_TIMESTAMP</td>
<td>SQL_CODE_TIMESTAMP</td>
</tr>
</tbody>
</table>

This field can be set for the interval data types listed in the following table.

<table>
<thead>
<tr>
<th>Interval type</th>
<th>DATETIME_INTERVAL_CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_INTERVAL_DAY/SQL_C_INTERVAL_DAY</td>
<td>SQL_CODE_DAY</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_HOUR/SQL_C_INTERVAL_DAY_TO_HOUR</td>
<td>SQL_CODE_DAY_TO_HOUR</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_MINUTE/SQL_C_INTERVAL_DAY_TO_MINUTE</td>
<td>SQL_CODE_DAY_TO_MINUTE</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_SECOND/</td>
<td>SQL_CODE_DAY_TO_SECOND</td>
</tr>
</tbody>
</table>
For more information about the data intervals and this field, see Data Type Identifiers and Descriptors.

**SQL_DESC_DATETIME_INTERVAL_PRECISION [All]**
This SQLINTEGER record field contains the interval leading precision if the SQL_DESC_TYPE field is SQL_INTERVAL. When the SQL_DESC_DATETIME_INTERVAL_CODE field is set to an interval data type, this field is set to the default interval leading precision.

**SQL_DESC_DISPLAY_SIZE [IRDs]**
This read-only SQLINTEGER record field contains the maximum number of characters required to display the data from the column.

**SQL_DESC_FIXED_PREC_SCALE [Implementation descriptors]**
This read-only SQLSMALLINT record field is set to SQL_TRUE if the column is an exact numeric column and has a fixed precision and nonzero scale, or to SQL_FALSE if the column is not an exact numeric column with a fixed precision and scale.

**SQL_DESC_INDICATOR_PTR [Application descriptors]**
In ARDs, this SQLLEN * record field points to the indicator variable. This variable contains SQL_NULL_DATA if the column value is a NULL. For APDs, the indicator variable is set to SQL_NULL_DATA to specify NULL dynamic arguments. Otherwise, the variable is zero (unless the values in SQL_DESC_INDICATOR_PTR and SQL_DESC_OCTET_LENGTH_PTR are the same pointer).

If the SQL_DESC_INDICATOR_PTR field in an ARD is a null pointer, the driver is prevented from returning information about whether the column is NULL or not. If the column is NULL and SQL_DESC_INDICATOR_PTR is a null pointer, SQLSTATE 22002 (Indicator variable required but not supplied) is returned when the driver attempts to populate the buffer after a call to SQLFetch or SQLFetchScroll. If the call to SQLFetch or SQLFetchScroll did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.
The `SQL_DESC_INDICATOR_PTR` field determines whether the field pointed to by `SQL_DESC_OCTET_LENGTH_PTR` is set. If the data value for a column is NULL, the driver sets the indicator variable to `SQL_NULL_DATA`. The field pointed to by `SQL_DESC_OCTET_LENGTH_PTR` is then not set. If a NULL value is not encountered during the fetch, the buffer pointed to by `SQL_DESC_INDICATOR_PTR` is set to zero and the buffer pointed to by `SQL_DESC_OCTET_LENGTH_PTR` is set to the length of the data.

If the `SQL_DESC_INDICATOR_PTR` field in an APD is a null pointer, the application cannot use this descriptor record to specify NULL arguments.

This field is a *deferred field*: It is not used at the time it is set but is used at a later time by the driver to indicate nullability (for ARDs) or to determine nullability (for APDs).

**SQL_DESC_LABEL [IRDs]**
This read-only `SQLCHAR *` record field contains the column label or title. If the column does not have a label, this variable contains the column name. If the column is unnamed and unlabeled, this variable contains an empty string.

**SQL_DESC_LENGTH [All]**
This `SQLULEN` record field is either the maximum or actual length of a character string in characters or a binary data type in bytes. It is the maximum length for a fixed-length data type, or the actual length for a variable-length data type. Its value always excludes the null-termination character that ends the character string. For values whose type is `SQL_TYPE_DATE`, `SQL_TYPE_TIME`, `SQL_TYPE_TIMESTAMP`, or one of the SQL interval data types, this field has the length in characters of the character string representation of the datetime or interval value.

The value in this field may be different from the value for "length" as defined in ODBC 2.x. For more information, see Appendix D: Data Types.

**SQL_DESC_LITERAL_PREFIX [IRDs]**
This read-only `SQLCHAR *` record field contains the character or characters that the driver recognizes as a prefix for a literal of this data type. This variable contains an empty string for a data type for which a literal prefix is not applicable.

**SQL_DESC_LITERAL_SUFFIX [IRDs]**
This read-only `SQLCHAR *` record field contains the character or characters that the driver recognizes as a suffix for a literal of this data type. This variable contains an empty string for a data type for which a literal suffix is not applicable.

**SQL_DESC_LOCAL_TYPE_NAME [Implementation descriptors]**
This read-only `SQLCHAR *` record field contains any localized (native language) name for the data type that may be different from the regular name of the data type. If there is no localized name, an empty string is returned. This field is for display purposes only.

**SQL_DESC_NAME [Implementation descriptors]**
This `SQLCHAR *` record field in a row descriptor contains the column alias, if it applies. If the column alias does not apply, the column name is returned. In either case, the driver sets the `SQL_DESC_UNNAMED` field to `SQL_NAMED` when it sets the `SQL_DESC_NAME` field. If there is no column name or a column alias, the driver returns an empty string in the `SQL_DESC_NAME` field and sets the `SQL_DESC_UNNAMED` field to `SQL_UNNAMED`.

An application can set the `SQL_DESC_NAME` field of an IPD to a parameter name or alias to specify stored procedure parameters by name. (For more information, see Binding Parameters by Name (Named Parameters).) The `SQL_DESC_NAME` field of an IRD is a read-only field; `SQLSTATE HY091` (Invalid descriptor field identifier) will be returned if an application attempts to set it.

In IPDs, this field is undefined if the driver does not support named parameters. If the driver
supports named parameters and is capable of describing parameters, the parameter name is returned in this field.

**SQL_DESC_NULLABLE [Implementation descriptors]**

In IRDs, this read-only SQLSMALLINT record field is SQL_NULLABLE if the column can have NULL values, SQL_NO_NULLS if the column does not have NULL values, or SQL_NULLABLE_UNKNOWN if it is not known whether the column accepts NULL values. This field pertains to the result set column, not the base column.

In IPDs, this field is always set to SQL_NULLABLE because dynamic parameters are always nullable and cannot be set by an application.

**SQL_DESC_NUM_PREC_RADIX [All]**

This SQLINTEGER field contains a value of 2 if the data type in the SQL_DESC_TYPE field is an approximate numeric data type, because the SQL_DESC_PRECISION field contains the number of bits. This field contains a value of 10 if the data type in the SQL_DESC_TYPE field is an exact numeric data type, because the SQL_DESC_PRECISION field contains the number of decimal digits. This field is set to 0 for all non-numeric data types.

**SQL_DESC_OCTET_LENGTH [All]**

This SQLLEN record field contains the length, in bytes, of a character string or binary data type. For fixed-length character or binary types, this is the actual length in bytes. For variable-length character or binary types, this is the maximum length in bytes. This value always excludes space for the null-termination character for implementation descriptors and always includes space for the null-termination character for application descriptors. For application data, this field contains the size of the buffer. For APDs, this field is defined only for output or input/output parameters.

**SQL_DESC_OCTET_LENGTH_PTR [Application descriptors]**

This SQLLEN * record field points to a variable that will contain the total length in bytes of a dynamic argument (for parameter descriptors) or of a bound column value (for row descriptors).

For an APD, this value is ignored for all arguments except character string and binary; if this field points to SQL_NTS, the dynamic argument must be null-terminated. To indicate that a bound parameter will be a data-at-execution parameter, an application sets this field in the appropriate record of the APD to a variable that, at execute time, will contain the value SQL_DATA_AT_EXEC or the result of the SQL_LEN_DATA_AT_EXEC macro. If there is more than one such field, SQL_DESC_DATA_PTR can be set to a value uniquely identifying the parameter to help the application determine which parameter is being requested.

If the OCTET_LENGTH_PTR field of an ARD is a null pointer, the driver does not return length information for the column. If the SQL_DESC_OCTET_LENGTH_PTR field of an APD is a null pointer, the driver assumes that character strings and binary values are null-terminated. (Binary values should not be null-terminated but should be given a length to avoid truncation.)

If the call to SQLFetch or SQLFetchScroll that fills in the buffer pointed to by this field did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined. This field is a deferred field. It is not used at the time it is set but is used at a later time by the driver to determine or indicate the octet length of the data.

**SQL_DESC_PARAMETER_TYPE [IPDs]**

This SQLSMALLINT record field is set to SQL_PARAM_INPUT for an input parameter, SQL_PARAM_INPUT_OUTPUT for an input/output parameter, SQL_PARAM_OUTPUT for an output parameter, SQL_PARAM_INPUT_OUTPUT_STREAM for an input/output streamed parameter, or SQL_PARAM_OUTPUT_STREAM for an output streamed parameter. It is set to SQL_PARAM_INPUT by default.

For an IPD, the field is set to SQL_PARAM_INPUT by default if the IPD is not automatically
populated by the driver (the SQL_ATTR_ENABLE_AUTO_IPD statement attribute is SQL_FALSE). An application should set this field in the IPD for parameters that are not input parameters.

**SQL_DESCPRECISION [All]**

This SQLSMALLINT record field contains the number of digits for an exact numeric type, the number of bits in the mantissa (binary precision) for an approximate numeric type, or the numbers of digits in the fractional seconds component for the SQL_TYPE_TIME, SQL_TYPE_TIMESTAMP, or SQL_INTERVAL_SECOND data type. This field is undefined for all other data types.

The value in this field may be different from the value for "precision" as defined in ODBC 2.x. For more information, see Appendix D: Data Types.

**SQL_DESC_ROWVER [Implementation descriptors]**

This SQLSMALLINT record field indicates whether a column is automatically modified by the DBMS when a row is updated (for example, a column of the type "timestamp" in SQL Server). The value of this record field is set to SQL_TRUE if the column is a row versioning column, and to SQL_FALSE otherwise. This column attribute is similar to calling SQLSpecialColumns with IdentifierType of SQL_ROWVER to determine whether a column is automatically updated.

**SQL_DESC_SCALE [All]**

This SQLSMALLINT record field contains the defined scale for decimal and numeric data types. The field is undefined for all other data types.

The value in this field may be different from the value for "scale" as defined in ODBC 2.x. For more information, see Appendix D: Data Types.

**SQL_DESC_SCHEMA_NAME [IRDs]**

This read-only SQLCHAR * record field contains the schema name of the base table that contains the column. The return value is driver-dependent if the column is an expression or if the column is part of a view. If the data source does not support schemas or the schema name cannot be determined, this variable contains an empty string.

**SQL_DESC_SEARCHABLE [IRDs]**

This read-only SQLSMALLINT record field is set to one of the following values:

- SQL_PRED_NONE if the column cannot be used in a WHERE clause. (This is the same as the SQL_UNSEARCHABLE value in ODBC 2.x.)
- SQL_PRED_CHAR if the column can be used in a WHERE clause but only with the LIKE predicate. (This is the same as the SQL_LIKE_ONLY value in ODBC 2.x.)
- SQL_PRED_BASIC if the column can be used in a WHERE clause with all the comparison operators except LIKE. (This is the same as the SQL_EXCEPT_LIKE value in ODBC 2.x.)
- SQL_PRED_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.

**SQL_DESC_TABLE_NAME [IRDs]**

This read-only SQLCHAR * record field contains the name of the base table that contains this column. The return value is driver-dependent if the column is an expression or if the column is part of a view.

**SQL_DESC_TYPE [All]**

This SQLSMALLINT record field specifies the concise SQL or C data type for all data types except datetime and interval data types. For the datetime and interval data types, this field specifies the verbose data type, which is SQL_DATETIME or SQL_INTERVAL.
Whenever this field contains SQL_DATETIME or SQL_INTERVAL, the SQL_DESC_DATETIME_INTERVAL_CODE field must contain the appropriate subcode for the concise type. For datetime data types, SQL_DESC_TYPE contains SQL_DATETIME, and the SQL_DESC_DATETIME_INTERVAL_CODE field contains a subcode for the specific datetime data type. For interval data types, SQL_DESC_TYPE contains SQL_INTERVAL and the SQL_DESC_DATETIME_INTERVAL_CODE field contains a subcode for the specific interval data type.

The values in the SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE fields are interdependent. Each time one of the fields is set, the other must also be set. SQL_DESC_TYPE can be set by a call to SQLSetDescField or SQLSetDescRec. SQL_DESC_CONCISE_TYPE can be set by a call to SQLBindCol or SQLBindParameter, or SQLSetDescField.

If SQL_DESC_TYPE is set to a concise data type other than an interval or datetime data type, the SQL_DESC_CONCISE_TYPE field is set to the same value and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to 0.

If SQL_DESC_TYPE is set to the verbose datetime or interval data type (SQL_DATETIME or SQL_INTERVAL) and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to the appropriate subcode, the SQL_DESC_CONCISE_TYPE field is set to the corresponding concise type. Trying to set SQL_DESC_TYPE to one of the concise datetime or interval types will return SQLSTATE HY021 (Inconsistent descriptor information).

When the SQL_DESC_TYPE field is set by a call to SQLBindCol, SQLBindParameter, or SQLSetDescField, the following fields are set to the following default values, as shown in the table below. The values of the remaining fields of the same record are undefined.

<table>
<thead>
<tr>
<th>Value of SQL_DESC_TYPE</th>
<th>Other fields implicitly set</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR, SQL_VARCHAR, SQL_C_CHAR, SQL_C_VARCHAR</td>
<td>SQL_DESC_LENGTH is set to 1. SQL_DESC_PRECISION is set to 0.</td>
</tr>
<tr>
<td>SQL_DATETIME</td>
<td>When SQL_DESC_DATETIME_INTERVAL_CODE is set to SQL_CODE_DATE or SQL_CODE_TIME, SQL_DESC_PRECISION is set to 0. When it is set to SQL_DESC_TIMESTAMP, SQL_DESC_PRECISION is set to 6.</td>
</tr>
<tr>
<td>SQL_DECIMAL, SQL_NUMERIC, SQL_C_NUMERIC</td>
<td>SQL_DESC_SCALE is set to 0. SQL_DESC_PRECISION is set to the implementation-defined precision for the respective data type. See SQL to C: Numeric for information on how to manually bind a SQL_C_NUMERIC value.</td>
</tr>
<tr>
<td>SQL_FLOAT, SQL_C_FLOAT</td>
<td>SQL_DESC_PRECISION is set to the implementation-defined default precision for SQL_FLOAT.</td>
</tr>
<tr>
<td>SQL_INTERVAL</td>
<td>When SQL_DESC_DATETIME_INTERVAL_CODE is set to an interval data type, SQL_DESC_DATETIME_INTERVAL_PRECISION is set to 2 (the default interval leading precision). When the interval has a seconds component, SQL_DESC_PRECISION is set to 6 (the default interval seconds precision).</td>
</tr>
</tbody>
</table>
When an application calls `SQLSetDescField` to set fields of a descriptor rather than calling `SQLSetDescRec`, the application must first declare the data type. When it does, the other fields indicated in the previous table are implicitly set. If any of the values implicitly set are unacceptable, the application can then call `SQLSetDescField` or `SQLSetDescRec` to set the unacceptable value explicitly.

**SQL_DESC_TYPE_NAME [Implementation descriptors]**
This read-only SQLCHAR * record field contains the data source–dependent type name (for example, "CHAR", "VARCHAR", and so on). If the data type name is unknown, this variable contains an empty string.

**SQL_DESC_UNNAMED [Implementation descriptors]**
This SQLSMALLINT record field in a row descriptor is set by the driver to either SQL_NAMED or SQL_UNNAMED when it sets the SQL_DESC_NAME field. If the SQL_DESC_NAME field contains a column alias or if the column alias does not apply, the driver sets the SQL_DESC_UNNAMED field to SQL_NAMED. If an application sets the SQL_DESC_NAME field of an IPD to a parameter name or alias, the driver sets the SQL_DESC_UNNAMED field of the IPD to SQL_NAMED. If there is no column name or a column alias, the driver sets the SQL_DESC_UNNAMED field to SQL_UNNAMED.

An application can set the SQL_DESC_UNNAMED field of an IPD to SQL_UNNAMED. A driver returns SQLSTATE HY091 (Invalid descriptor field identifier) if an application attempts to set the SQL_DESC_UNNAMED field of an IPD to SQL_NAMED. The SQL_DESC_UNNAMED field of an IRD is read-only; SQLSTATE HY091 (Invalid descriptor field identifier) will be returned if an application attempts to set it.

**SQL_DESC_UNSIGNED [Implementation descriptors]**
This read-only SQLSMALLINT record field is set to SQL_TRUE if the column type is unsigned or non-numeric, or SQL_FALSE if the column type is signed.

**SQL_DESC_UPDATABLE [IRDs]**
This read-only SQLSMALLINT record field is set to one of the following values:

- SQL_ATTR_READ_ONLY if the result set column is read-only.
- SQL_ATTR_WRITE if the result set column is read-write.
- SQL_ATTR_READWRITE_UNKNOWN if it is not known whether the result set column is updatable or not.

SQL_DESC_UPDATABLE describes the updatability of the column in the result set, not the column in the base table. The updatability of the column in the base table on which this result set column is based may be different than the value in this field. Whether a column is updatable can be based on the data type, user privileges, and the definition of the result set itself. If it is unclear whether a column is updatable, SQL_ATTR_READWRITE_UNKNOWN should be returned.

**Consistency Checks**

A consistency check is performed by the driver automatically whenever an application passes in a value for the SQL_DESC_DATA_PTR field of the ARD, APD, or IPD. If any of the fields is inconsistent with other fields, `SQLSetDescField` will return SQLSTATE HY021 (Inconsistent descriptor information). For more information, see "Consistency Check" in `SQLSetDescRec`. 
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a column</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Binding a parameter</td>
<td>SQLBindParameter Function</td>
</tr>
<tr>
<td>Getting a descriptor field</td>
<td>SQLGetDescField Function</td>
</tr>
<tr>
<td>Getting multiple descriptor fields</td>
<td>SQLGetDescRec Function</td>
</tr>
<tr>
<td>Setting multiple descriptor fields</td>
<td>SQLSetDescRec Function</td>
</tr>
</tbody>
</table>

See Also

ODBC Header Files
ODBC API Reference

© 2016 Microsoft

SQLSetDescRec Function

Conformance

Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary

The **SQLSetDescRec** function sets multiple descriptor fields that affect the data type and buffer bound to a column or parameter data.

Syntax

```c
SQLRETURN SQLSetDescRec(
    SQLHDESC    DescriptorHandle,
    SQLSMALLINT RecNumber,
    SQLSMALLINT Type,
    SQLSMALLINT SubType,
    SQLLEN      Length,
    SQLSMALLINT Precision,
    SQLSMALLINT Scale,
    SQLPOINTER  DataPtr,
);```

```c
```
Arguments

DescriptorHandle
[Input] Descriptor handle. This must not be an IRD handle.

RecNumber
[Input] Indicates the descriptor record that contains the fields to be set. Descriptor records are numbered from 0, with record number 0 being the bookmark record. This argument must be equal to or greater than 0. If RecNumber is greater than the value of SQL_DESC_COUNT, SQL_DESC_COUNT is changed to the value of RecNumber.

Type
[Input] The value to which to set the SQL_DESC_TYPE field for the descriptor record.

SubType
[Input] For records whose type is SQL_DATETIME or SQL_INTERVAL, this is the value to which to set the SQL_DESC_DATETIME_INTERVAL_CODE field.

Length
[Input] The value to which to set the SQL_DESC_OCTET_LENGTH field for the descriptor record.

Precision
[Input] The value to which to set the SQL_DESC_PRECISION field for the descriptor record.

Scale
[Input] The value to which to set the SQL_DESC_SCALE field for the descriptor record.

DataPtr
[Deferred Input or Output] The value to which to set the SQL_DESC_DATA_PTR field for the descriptor record. DataPtr can be set to a null pointer.

The DataPtr argument can be set to a null pointer to set the SQL_DESC_DATA_PTR field to a null pointer. If the handle in the DescriptorHandle argument is associated with an ARD, this unbinds the column.

StringLengthPtr
[Deferred Input or Output] The value to which to set the SQL_DESC_OCTET_LENGTH_PTR field for the descriptor record. StringLengthPtr can be set to a null pointer.

IndicatorPtr
[Deferred Input or Output] The value to which to set the SQL_DESC_INDEX_PTR field for the descriptor record. IndicatorPtr can be set to a null pointer.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.
Diagnostics

When [SQLSetDescRec](https://www.sql.com/sqlsetdescrec) returns `SQL_ERROR` or `SQL_SUCCESS_WITH_INFO`, an associated SQLSTATE value can be obtained by calling [SQLGetDiagRec](https://www.sql.com/sqlgetdiagrec) with a [HandleType](https://www.sql.com/handletype) of `SQL_HANDLE_DESC` and a [Handle](https://www.sql.com/handle) of [DescriptorHandle](https://www.sql.com/descr). The following table lists the SQLSTATE values commonly returned by [SQLSetDescRec](https://www.sql.com/sqlsetdescrec) and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is `SQL_ERROR`, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns <code>SQL_SUCCESS_WITH_INFO</code>.)</td>
</tr>
</tbody>
</table>
| 07009    | Invalid descriptor index | The `RecNumber` argument was set to 0, and the [DescriptorHandle](https://www.sql.com/descr) referred to an IPD handle.  

  The `RecNumber` argument was less than 0.  

  The `RecNumber` argument was greater than the maximum number of columns or parameters that the data source can support, and the [DescriptorHandle](https://www.sql.com/descr) argument was an APD, IPD, or ARD.  

  The `RecNumber` argument was equal to 0, and the [DescriptorHandle](https://www.sql.com/descr) argument referred to an implicitly allocated APD. (This error does not occur with an explicitly allocated application descriptor because it is not known whether an explicitly allocated application descriptor is an APD or ARD until execute time.) |
| 08S01    | Communication link failure | The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.                                                               |
| HY000    | General error          | An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by [SQLGetDiagRec](https://www.sql.com/sqlgetdiagrec) in the *MessageText* buffer describes the error and its cause. |
| HY001    | Memory allocation error | The driver was unable to allocate memory required to support execution or completion of the function.                                                                                                  |
| HY010    | Function sequence error | (DM) The [DescriptorHandle](https://www.sql.com/descr) was associated with a [StatementHandle](https://www.sql.com/statement) for which an asynchronously executing function (not this one) was called and was still executing when this function was called.  

  (DM) [SQLExecute](https://www.sql.com/sqlexecute), [SQLExecDirect](https://www.sqlcom/sqlexecdirect), [SQLBulkOperations](https://www.sqlcom/sqlbulkoperations), or [SQLSetPos](https://www.sqlcom/sqlsetpos) was called for the [StatementHandle](https://www.sqlcom/statement) with which the [DescriptorHandle](https://www.sqlcom/descr) was associated and returned `SQL_NEED_DATA`. This function was called before data was sent for all data-at-execution parameters or columns. |
(DM) An asynchronously executing function was called for the connection handle that is associated with the DescriptorHandle. This asynchronous function was still executing when the SQLSetDescRec function was called.

(DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for one of the statement handles associated with the DescriptorHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters.

<table>
<thead>
<tr>
<th>HY013</th>
<th>Memory management error</th>
<th>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY016</td>
<td>Cannot modify an implementation row descriptor</td>
<td>The DescriptorHandle argument was associated with an IRD.</td>
</tr>
<tr>
<td>HY021</td>
<td>Inconsistent descriptor information</td>
<td>The Type field, or any other field associated with the SQL_DESC_TYPE field in the descriptor, was not valid or consistent. Descriptor information checked during a consistency check was not consistent. (See “Consistency Checks,” later in this section.)</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The driver was an ODBC 2.x driver, the descriptor was an ARD, the ColumnNumber argument was set to 0, and the value specified for the argument BufferLength was not equal to 4.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the DescriptorHandle does not support the function.</td>
</tr>
</tbody>
</table>

**Comments**

An application can call SQLSetDescRec to set the following fields for a single column or parameter:
• SQL_DESC_TYPE
• SQL_DESC_DATETIME_INTERVAL_CODE (for records whose type is SQL_DATETIME or SQL_INTERVAL)
• SQL_DESC_OCTET_LENGTH
• SQL_DESC_PRECISION
• SQL_DESC_SCALE
• SQL_DESC_DATA_PTR
• SQL_DESC_OCTET_LENGTH_PTR
• SQL_DESC_INDICATOR_PTR

Note

If a call to SQLSetDescRec fails, the contents of the descriptor record identified by the RecNumber argument are undefined.

When binding a column or parameter, SQLSetDescRec allows you to change multiple fields affecting the binding without calling SQLBindCol or SQLBindParameter or making multiple calls to SQLSetDescField. SQLSetDescRec can set fields on a descriptor not currently associated with a statement. Note that SQLBindParameter sets more fields than SQLSetDescRec, can set fields on both an APD and an IPD in one call, and does not require a descriptor handle.

Note

The statement attribute SQL_ATTR_USE_BOOKMARKS should always be set before calling SQLSetDescRec with a RecNumber argument of 0 to set bookmark fields. While this is not mandatory, it is strongly recommended.

Consistency Checks

A consistency check is performed by the driver automatically whenever an application sets the SQL_DESC_DATA_PTR field of an APD, ARD, or IPD. If any of the fields is inconsistent with other fields, SQLSetDescRec will return SQLSTATE HY021 (Inconsistent descriptor information).

Whenever an application sets the SQL_DESC_DATA_PTR field of an APD, ARD, or IPD, the driver checks that the value of the SQL_DESC_TYPE field and the values applicable to that SQL_DESC_TYPE field are valid and consistent. This check is always performed when SQLBindParameter or SQLBindCol is called or when SQLSetDescRec is called for an APD, ARD, or IPD. This consistency check includes the following checks on descriptor fields:

• The SQL_DESC_TYPE field must be one of the valid ODBC C or SQL types or a driver-specific SQL type. The SQL_DESC_CONCISE_TYPE field must be one of the valid ODBC C or SQL types or a driver-specific C or SQL type, including the concise datetime and interval types.

• If the SQL_DESC_TYPE record field is SQL_DATETIME or SQL_INTERVAL, the SQL_DESC_DATETIME_INTERVAL_CODE field must be one of the valid datetime or interval types.
codes. (See the description of the SQL_DESC_DATETIME_INTERVAL_CODE field in SQLSetDescField.)

- If the SQL_DESC_TYPE field indicates a numeric type, the SQL_DESC_PRECISION and SQL_DESC_SCALE fields are verified to be valid.

- If the SQL_DESC_CONCISE_TYPE field is a time or timestamp data type, an interval type with a seconds component, or one of the interval data types with a time component, the SQL_DESC_PRECISION field is verified to be a valid seconds precision.

- If the SQL_DESC_CONCISE_TYPE is an interval data type, the SQL_DESC_DATETIME_INTERVAL_PRECISION field is verified to be a valid interval leading precision value.

The SQL_DESC_DATA_PTR field of an IPD is not normally set; however, an application can do so to force a consistency check of IPD fields. A consistency check cannot be performed on an IRD. The value that the SQL_DESC_DATA_PTR field of the IPD is set to is not actually stored and cannot be retrieved by a call to SQLGetDescField or SQLGetDescRec; the setting is made only to force the consistency check.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a column</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Binding a parameter</td>
<td>SQLBindParameter Function</td>
</tr>
<tr>
<td>Getting a single descriptor field</td>
<td>SQLGetDescField Function</td>
</tr>
<tr>
<td>Getting multiple descriptor fields</td>
<td>SQLGetDescRec Function</td>
</tr>
<tr>
<td>Setting single descriptor fields</td>
<td>SQLSetDescField Function</td>
</tr>
</tbody>
</table>

**See Also**

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

**SQLSetEnvAttr Function**

**Conformance**
Summary

`SQLSetEnvAttr` sets attributes that govern aspects of environments.

Syntax

```c
SQLRETURN SQLSetEnvAttr(
    SQLHENV EnvironmentHandle,
    SQLINTEGER Attribute,
    SQLPOINTER ValuePtr,
    SQLINTEGER StringLength);
```

Arguments

- **EnvironmentHandle**
  
  [Input] Environment handle.

- **Attribute**
  
  [Input] Attribute to set, listed in "Comments."

- **ValuePtr**
  
  [Input] Pointer to the value to be associated with `Attribute`. Depending on the value of `Attribute`, `ValuePtr` will be a 32-bit integer value or point to a null-terminated character string.

- **StringLength**
  
  [Input] If `ValuePtr` points to a character string or a binary buffer, this argument should be the length of `*ValuePtr`. For character string data, this argument should contain the number of bytes in the string.

  If `ValuePtr` is an integer, `StringLength` is ignored.

Returns

`SQL_SUCCESS`, `SQL_SUCCESS_WITH_INFO`, `SQL_ERROR`, or `SQL_INVALID_HANDLE`.

Diagnostics

When `SQLSetEnvAttr` returns `SQL_ERROR` or `SQL_SUCCESS_WITH_INFO`, an associated SQLSTATE value can be obtained by calling `SQLGetDiagRec` with a `HandleType` of `SQL_HANDLE_ENV` and a `Handle` of `EnvironmentHandle`. The following table lists the SQLSTATE values typically returned by `SQLSetEnvAttr` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is `SQL_ERROR`, unless noted otherwise. If a driver does not support an environment attribute, the error can be returned only during connect time.
<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed</td>
<td>The driver did not support the value specified in ValuePtr and substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>The Attribute argument identified an environment attribute that required a string value, and the ValuePtr argument was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) A connection handle has been allocated on EnvironmentHandle. (DM) SQL_ATTR_ODBC_VERSION has not been set with SQLSetEnvAttr and Attribute is not equal to SQL_ATTR_ODBC_VERSION. You do not need to set SQL_ATTR_ODBC_VERSION explicitly if you are using SQLAllocHandleStd.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY024</td>
<td>Invalid attribute value</td>
<td>Given the specified Attribute value, an invalid value was specified in ValuePtr.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>The StringLength argument was less than 0 but was not SQL_NTS.</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute/option identifier</td>
<td>(DM) The value specified for the argument Attribute was not valid for the version of ODBC supported by the driver.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>The value specified for the argument Attribute was a valid ODBC environment attribute for the version of ODBC supported by the driver, but was not supported by</td>
</tr>
</tbody>
</table>
An application can call SQLSetEnvAttr only if no connection handle is allocated on the environment. All environment attributes successfully set by the application for the environment persist until SQLFreeHandle is called on the environment. More than one environment handle can be allocated simultaneously in ODBC 3.x.

The format of information set through ValuePtr depends on the specified Attribute. SQLSetEnvAttr will accept attribute information in one of two different formats: a null-terminated character string or a 32-bit integer value. The format of each is noted in the attribute’s description.

There are no driver-specific environment attributes.

Connection attributes cannot be set by a call to SQLSetEnvAttr. Trying to do this will return SQLSTATE HY092 (Invalid attribute/option identifier).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>ValuePtr contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_CONNECTION_POOLING (ODBC 3.8)</td>
<td>A 32-bit SQLUINTEGER value that enables or disables connection pooling at the environment level. The following values are used:</td>
</tr>
<tr>
<td>SQL_CP_OFF = Connection pooling is turned off. This is the default.</td>
<td></td>
</tr>
<tr>
<td>SQL_CP_ONE_PER_DRIVER = A single connection pool is supported for each driver. Every connection in a pool is associated with one driver.</td>
<td></td>
</tr>
<tr>
<td>SQL_CP_ONE_PER_HENV = A single connection pool is supported for each environment. Every connection in a pool is associated with one environment.</td>
<td></td>
</tr>
<tr>
<td>SQL_CP_DRIVER_AWARE = Use the connection-pool awareness feature of the driver, if it is available. If the driver does not support connection-pool awareness, SQL_CP_DRIVER_AWARE is ignored and SQL_CP_ONE_PER_HENV is used. For more information, see Driver-Aware Connection Pooling. In an environment where some drivers support and some drivers do not support connection-pool awareness, SQL_CP_DRIVER_AWARE can enable the connection-pool awareness feature on those supporting drivers, but it is equivalent to setting to SQL_CP_ONE_PER_HENV on those drivers that do not support connection-pool awareness feature.</td>
<td></td>
</tr>
</tbody>
</table>

Connection pooling is enabled by calling SQLSetEnvAttr to set the SQL_ATTR_CONNECTION_POOLING attribute to...
SQL_CP_ONE_PER_DRIVER or SQL_CP_ONE_PER_HENV. This call must be made before the application allocates the shared environment for which connection pooling is to be enabled. The environment handle in the call to SQLSetEnvAttr is set to null, which makes SQL_ATTR_CONNECTION_POOLING a process-level attribute. After connection pooling is enabled, the application then allocates an implicit shared environment by calling SQLAllocHandle with the InputHandle argument set to SQL_HANDLE_ENV.

After connection pooling has been enabled and a shared environment has been selected for an application, SQL_ATTR_CONNECTION_POOLING cannot be reset for that environment, because SQLSetEnvAttr is called with a null environment handle when setting this attribute. If this attribute is set while connection pooling is already enabled on a shared environment, the attribute affects only shared environments that are allocated subsequently.

It is also possible to enable connection pooling on an environment. Note the following about environment connection pooling:

- Enabling connection pooling on a NULL handle is a process-level attribute. Subsequently allocated environments will be a shared environment, and will inherit the process-level connection pooling setting.
- After an environment is allocated, an application can still change its connection pool setting.
- If environment connection pooling is enabled and the connection's driver uses driver pooling, environment pooling takes preference.

SQL_ATTR_CONNECTION_POOLING is implemented inside the Driver Manager. A driver does not need to implement SQL_ATTR_CONNECTION_POOLING. ODBC 2.0 and 3.0 applications can set this environment attribute.

For more information, see ODBC Connection Pooling.

**SQL_ATTR_CP_MATCH (ODBC 3.0)**

A 32-bit SQLUINTEGER value that determines how a connection is chosen from a connection pool. When SQLConnect or SQLDriverConnect is called, the Driver Manager determines which connection is reused from the pool. The Driver Manager tries to match the connection options in the call and the connection attributes set by the application to the keywords and connection attributes of the connections in the pool. The value of this attribute determines the level of precision of the matching criteria.

The following values are used to set the value of this attribute:

SQL_CP_STRICT_MATCH = Only connections that exactly match the connection options in the call and the connection attributes set by the application are reused. This is the
SQL_CP_RELAXED_MATCH = Connections with matching connection string keywords can be used. Keywords must match, but not all connection attributes must match.

For more information about how the Driver Manager performs the match in connecting to a pooled connection, see SQLConnect. For more information about connection pooling, see ODBC Connection Pooling.

SQL_ATTR_ODBC_VERSION (ODBC 3.0)

A 32-bit integer that determines whether certain functionality exhibits ODBC 2.x behavior or ODBC 3.x behavior. The following values are used to set the value of this attribute:

SQL_OV_ODBC3_80 = The Driver Manager and driver exhibit the following ODBC 3.8 behavior:

- The driver returns and expects ODBC 3.x codes for date, time, and timestamp.
- The driver returns ODBC 3.x SQLSTATE codes when SQLError, SQLGetDiagField, or SQLGetDiagRec is called.
- The CatalogName argument in a call to SQLTables accepts a search pattern.
- The Driver Manager supports C data type extensibility. For more information about C data type extensibility, see C Data Types in ODBC.

For more information, see What's New in ODBC 3.8.

SQL_OV_ODBC3 = The Driver Manager and driver exhibit the following ODBC 3.x behavior:

- The driver returns and expects ODBC 3.x codes for date, time, and timestamp.
- The driver returns ODBC 3.x SQLSTATE codes when SQLError, SQLGetDiagField, or SQLGetDiagRec is called.
- The CatalogName argument in a call to SQLTables accepts a search pattern.
- The Driver Manager does not support C data type extensibility.

SQL_OV_ODBC2 = The Driver Manager and driver exhibit the following ODBC 2.x behavior. This is especially useful for an ODBC 2.x application working with an ODBC 3.x driver.

- The driver returns and expects ODBC 2.x codes for date, time, and timestamp.
- The driver returns ODBC 2.x SQLSTATE codes when SQLError, SQLGetDiagField, or SQLGetDiagRec is called.
- The CatalogName argument in a call to SQLTables does not accept a search pattern.
The Driver Manager does not support C data type extensibility.

An application must set this environment attribute before it calls any function that has an SQLHENV argument, or the call will return SQLSTATE HY010 (Function sequence error). It is driver-specific whether additional behavior exists for these environmental flags.

- For more information, see Declaring the Application's ODBC Version and Behavioral Changes.

### SQL_ATTR_OUTPUT_NTS (ODBC 3.0)

A 32-bit integer that determines how the driver returns string data. If SQL_TRUE, the driver returns string data null-terminated. If SQL_FALSE, the driver does not return string data null-terminated.

This attribute defaults to SQL_TRUE. A call to SQLSetEnvAttr to set it to SQL_TRUE returns SQL_SUCCESS. A call to SQLSetEnvAttr to set it to SQL_FALSE returns SQL_ERROR and SQLSTATE HYC00 (Optional feature not implemented).

#### Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocating a handle</td>
<td>SQLAllocHandle Function</td>
</tr>
<tr>
<td>Returning the setting of an environment attribute</td>
<td>SQLGetEnvAttr Function</td>
</tr>
</tbody>
</table>

#### See Also

- ODBC API Reference
- ODBC Header Files
- What's New in ODBC 3.8

© 2016 Microsoft

**SQLSetParam Function**

**Conformance**

Version Introduced: ODBC 1.0 Standards Compliance: Deprecated
Summary

In ODBC 2.0, the ODBC 1.0 function `SQLSetParam` has been replaced by `SQLBindParameter`. For more information, see `SQLBindParameter Function`.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
</table>
| For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see `Mapping Deprecated Functions`.

Remarks

See `ODBC 64-Bit Information`, if your application will run on a 64-bit operating system.

See Also

- ODBC API Reference
- ODBC Header Files

© 2016 Microsoft

**SQLSetPos Function**

Conformance

Version Introduced: ODBC 1.0 Standards Compliance: ODBC

Summary

`SQLSetPos` sets the cursor position in a rowset and allows an application to refresh data in the rowset or to update or delete data in the result set.

Syntax

```c
SQLRETURN SQLSetPos(
    SQLHSTMT StatementHandle,
    SQLSETPOSIROW RowNumber,
    SQLUSMALLINT Operation,
    SQLUSMALLINT LockType);
```

Arguments

- **StatementHandle**
  - [Input] Statement handle.
RowNumber

[Input] Position of the row in the rowset on which to perform the operation specified with the Operation argument. If RowNumber is 0, the operation applies to every row in the rowset.

For additional information, see "Comments."

Operation

[Input] Operation to perform:

SQL_POSITION SQL_REFRESH SQL_UPDATE SQL_DELETE

Note

The SQL_ADD value for the Operation argument has been deprecated for ODBC 3.x. ODBC 3.x drivers will need to support SQL_ADD for backward compatibility. This functionality has been replaced by a call to SQLBulkOperations with an Operation of SQL_ADD. When an ODBC 3.x application works with an ODBC 2.x driver, the Driver Manager maps a call to SQLBulkOperations with an Operation of SQL_ADD to SQLSetPos with an Operation of SQL_ADD.

For more information, see "Comments."

LockType

[Input] Specifies how to lock the row after performing the operation specified in the Operation argument.

SQL_LOCK_NO_CHANGE SQL_LOCK_EXCLUSIVE SQL_LOCK_UNLOCK

For more information, see "Comments."

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLSetPos returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLSetPos and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

For all those SQLSTATEs that can return SQL_SUCCESS_WITH_INFO or SQL_ERROR (except 01xxx SQLSTATEs), SQL_SUCCESS_WITH_INFO is returned if an error occurs on one or more, but not all, rows of a multirow operation, and SQL_ERROR is returned if an error occurs on a single-row operation.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>Cursor operation conflict&lt;br&gt;The <em>Operation</em> argument was SQL_DELETE or SQL_UPDATE, and no rows or more than one row were deleted or updated. (For more information about updates to more than one row, see the description of the SQL_ATTR_SIMULATE_CURSOR Attribute in SQLSetStmtAttr.) (Function returns SQL_SUCCESS_WITH_INFO.)&lt;br&gt;The <em>Operation</em> argument was SQL_DELETE or SQL_UPDATE, and the operation failed because of optimistic concurrency. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
<td></td>
</tr>
<tr>
<td>01004</td>
<td>String data right truncation&lt;br&gt;The <em>Operation</em> argument was SQL_REFRESH, and string or binary data returned for a column or columns with a data type of SQL_C_CHAR or SQL_C_BINARY resulted in the truncation of nonblank character or non-NULL binary data.</td>
<td></td>
</tr>
<tr>
<td>01S01</td>
<td>Error in row&lt;br&gt;The <em>RowNumber</em> argument was 0, and an error occurred in one or more rows while performing the operation specified with the <em>Operation</em> argument. (SQL_SUCCESS_WITH_INFO is returned if an error occurs on one or more, but not all, rows of a multirow operation, and SQL_ERROR is returned if an error occurs on a single-row operation.) (This SQLSTATE is returned only when SQLSetPos is called after SQLExtendedFetch, if the driver is an ODBC 2.x driver and the cursor library is not used.)</td>
<td></td>
</tr>
<tr>
<td>01S07</td>
<td>Fractional truncation&lt;br&gt;The <em>Operation</em> argument was SQL_REFRESH, the data type of the application buffer was not SQL_C_CHAR or SQL_C_BINARY, and the data returned to application buffers for one or more columns was truncated. For numeric data types, the fractional part of the number was truncated. For time, timestamp, and interval data types containing a time component, the fractional portion of the time was truncated. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
<td></td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation&lt;br&gt;The data value of a column in the result set could not be converted to the data type specified by <em>TargetType</em> in the call to SQLBindCol.</td>
<td></td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index&lt;br&gt;The argument <em>Operation</em> was SQL_REFRESH or SQL_UPDATE, and a column was bound with a column number greater than the number of columns in the result set.</td>
<td></td>
</tr>
<tr>
<td>21S02</td>
<td>Degree of derived table does not match column list&lt;br&gt;The argument <em>Operation</em> was SQL_UPDATE, and no columns were updatable because all columns were either unbound, read-only, or the value in the bound length/indicator buffer was SQL_COLUMN_IGNORE.</td>
<td></td>
</tr>
<tr>
<td>22001</td>
<td>String data, right truncation&lt;br&gt;The <em>Operation</em> argument was SQL_UPDATE, and the assignment of a character or binary value to a column resulted in the truncation of nonblank (for characters) or non-</td>
<td></td>
</tr>
</tbody>
</table>
null (for binary) characters or bytes.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
</table>
| 22003 | Numeric value out of range         | The argument *Operation* was SQL_UPDATE, and the assignment of a numeric value to a column in the result set caused the whole (as opposed to fractional) part of the number to be truncated.  

The argument *Operation* was SQL_REFRESH, and returning the numeric value for one or more bound columns would have caused a loss of significant digits. |
| 22007 | Invalid datetime format            | The argument *Operation* was SQL_UPDATE, and the assignment of a date or timestamp value to a column in the result set caused the year, month, or day field to be out of range. 

The argument *Operation* was SQL_REFRESH, and returning the date or timestamp value for one or more bound columns would have caused the year, month, or day field to be out of range. |
| 22008 | Date/time field overflow           | The *Operation* argument was SQL_UPDATE, and the performance of datetime arithmetic on data being sent to a column in the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the Gregorian calendar's natural rules for datetimes. 

The *Operation* argument was SQL_REFRESH, and the performance of datetime arithmetic on data being retrieved from the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result being outside the permissible range of values for the field, or being invalid based on the Gregorian calendar's natural rules for datetimes. |
| 22015 | Interval field overflow            | The *Operation* argument was SQL_UPDATE, and the assignment of an exact numeric or interval C type to an interval SQL data type caused a loss of significant digits. 

The *Operation* argument was SQL_UPDATE; when assigning to an interval SQL type, there was no representation of the value of the C type in the interval SQL type. 

The *Operation* argument was SQL_REFRESH, and assigning from an exact numeric or interval SQL type to an interval C type caused a loss of significant digits in the leading field. 

The *Operation* argument was SQL_REFRESH; when assigning to an interval C type, there was no representation of the value of the SQL type in the interval C type. |
<p>| 22018 | Invalid character value for cast   | The <em>Operation</em> argument was SQL_REFRESH; the C type was an exact or approximate numeric, a datetime, or an interval |</p>
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>23000</td>
<td>Integrity constraint violation</td>
<td>The argument <em>Operation</em> was SQL_DELETE or SQL_UPDATE, and an integrity constraint was violated.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>The <em>StatementHandle</em> was in an executed state, but no result set was associated with the <em>StatementHandle</em>. (DM) A cursor was open on the <em>StatementHandle</em>, but SQLFetch or SQLFetchScroll had not been called. A cursor was open on the <em>StatementHandle</em>, and SQLFetch or SQLFetchScroll had been called, but the cursor was positioned before the start of the result set or after the end of the result set. The argument <em>Operation</em> was SQL_DELETE, SQL_REFRESH, or SQL_UPDATE, and the cursor was positioned before the start of the result set or after the end of the result set.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back due to a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>42000</td>
<td>Syntax error or access violation</td>
<td>The driver was unable to lock the row as needed to perform the operation requested in the argument <em>Operation</em>. The driver was unable to lock the row as requested in the argument <em>LockType</em>.</td>
</tr>
<tr>
<td>44000</td>
<td>WITH CHECK OPTION violation</td>
<td>The <em>Operation</em> argument was SQL_UPDATE, and the update was performed on a viewed table or a table derived from the viewed table which was created by specifying WITH CHECK OPTION, such that one or more rows affected by the update will no longer be present in the viewed table.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the <em>MessageText</em> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the StatementHandle. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle, and then the function was called again on the StatementHandle. The function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle from a different thread in a multithread application.</td>
</tr>
</tbody>
</table>
| HY010  | Function sequence error            | (DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLSetPos function was called.  
(DM) The specified StatementHandle was not in an executed state. The function was called without first calling SQLExecDirect, SQLExecute, or a catalog function.  
(DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called.  
(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.  
(DM) The driver was an ODBC 2.x driver, and SQLSetPos was called for a StatementHandle after SQLFetch was called. |
| HY011  | Attribute cannot be set now        | (DM) The driver was an ODBC 2.x driver; the SQL_ATTR_ROW_STATUS_PTR statement attribute was set; then SQLSetPos was called before SQLFetch, SQLFetchScroll, or SQLExtendedFetch was called. |
| HY013  | Memory management error            | The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions. |
| HY090  | Invalid string or buffer length    | The Operation argument was SQL_UPDATE, a data value was a null pointer, and the column length value was not 0, SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.  
The Operation argument was SQL_UPDATE; a data value was not a null pointer; the C data type was SQL_C_BINARY or SQL_C_CHAR; and the column length value was less than 0 but not equal to SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NTS, or SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.  
The value in a length/indicator buffer was |
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY092</td>
<td>Invalid attribute identifier</td>
<td>(DM) The value specified for the <em>Operation</em> argument was invalid. (DM) The value specified for the <em>LockType</em> argument was invalid. The <em>Operation</em> argument was SQL_UPDATE or SQL_DELETE, and the SQL_ATTR_CONCURRENCY statement attribute was SQL_ATTR_CONCUR_READ_ONLY.</td>
</tr>
<tr>
<td>HY107</td>
<td>Row value out of range</td>
<td>The value specified for the argument <em>RowNumber</em> was greater than the number of rows in the rowset.</td>
</tr>
</tbody>
</table>
| HY109 | Invalid cursor position                                                    | The cursor associated with the *StatementHandle* was defined as forward-only, so the cursor could not be positioned within the rowset. See the description for the SQL_ATTR_CURSOR_TYPE attribute in SQLSetStmtAttr.  

The *Operation* argument was SQL_UPDATE, SQL_DELETE, or SQL_REFRESH, and the row identified by the *RowNumber* argument had been deleted or had not been fetched. (DM) The *RowNumber* argument was 0, and the *Operation* argument was SQL_POSITION.  

SQLSetPos was called after SQLBulkOperations was called and before SQLFetchScroll or SQLFetch was called. |
| HY117 | Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed. | (DM) For more information about suspended state, see SQLEndTran Function.                                                                                                                                                                                                                                                                                   |
| HYC00 | Optional feature not implemented                                           | The driver or data source does not support the operation requested in the *Operation* argument or the *LockType* argument.                                                                                                                                                                                                                                   |
| HYT00 | Timeout expired                                                            | The query timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr with an Attribute of SQL_ATTR_QUERY_TIMEOUT.                                                                                                                                                                           |
| HYT01 | Connection timeout expired                                                 | The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.                                                                                                                                         |
**IM001**  
Driver does not support this function  
(DM) The driver associated with the *StatementHandle* does not support the function.

| IM017 | Polling is disabled in asynchronous notification mode | Whenever the notification model is used, polling is disabled. |
| IM018 | **SQLCompleteAsync** has not been called to complete the previous asynchronous operation on this handle. | If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, **SQLCompleteAsync** must be called on the handle to do post-processing and complete the operation. |

**Comments**

⚠️ **Caution**

For information on the statement states that **SQLSetPos** can be called in and what it needs to do for compatibility with ODBC 2.x applications, see Block Cursors, Scrollable Cursors, and Backward Compatibility.

**RowNumber Argument**

The *RowNumber* argument specifies the number of the row in the rowset on which to perform the operation specified by the *Operation* argument. If *RowNumber* is 0, the operation applies to every row in the rowset. *RowNumber* must be a value from 0 to the number of rows in the rowset.

⚠️ **Note**

In the C language, arrays are 0-based and the *RowNumber* argument is 1-based. For example, to update the fifth row of the rowset, an application modifies the rowset buffers at array index 4 but specifies a *RowNumber* of 5.

All operations position the cursor on the row specified by *RowNumber*. The following operations require a cursor position:

- Positioned update and delete statements.
- Calls to **SQLGetData**.
- Calls to **SQLSetPos** with the SQL_DELETE, SQL_REFRESH, and SQL_UPDATE options.

For example, if *RowNumber* is 2 for a call to **SQLSetPos** with an *Operation* of SQL_DELETE, the cursor is positioned on the second row of the rowset and that row is deleted. The entry in the implementation row status array (pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute) for the second row is changed to SQL_ROW_DELETED.
An application can specify a cursor position when it calls `SQLSetPos`. Generally, it calls `SQLSetPos` with the SQL_POSITION or SQL_REFRESH operation to position the cursor before executing a positioned update or delete statement or calling `SQLGetData`.

### Operation Argument

The *Operation* argument supports the following operations. To determine which options are supported by a data source, an application calls `SQLGetInfo` with the SQL_DYNAMIC_CURSOR_ATTRIBUTES1, SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1, SQL_KEYSET_CURSOR_ATTRIBUTES1, or SQL_STATIC_CURSOR_ATTRIBUTES1 information type (depending on the type of the cursor).

<table>
<thead>
<tr>
<th>Operation argument</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_POSITION</td>
<td>The driver positions the cursor on the row specified by <em>RowNumber</em>. The contents of the row status array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute are ignored for the SQL_POSITION Operation.</td>
</tr>
<tr>
<td>SQL_REFRESH</td>
<td>The driver positions the cursor on the row specified by <em>RowNumber</em> and refreshes data in the rowset buffers for that row. For more information about how the driver returns data in the rowset buffers, see the descriptions of row-wise and column-wise binding in <code>SQLBindCol</code>. <code>SQLSetPos</code> with an Operation of SQL_REFRESH updates the status and content of the rows within the current fetched rowset. This includes refreshing the bookmarks. Because the data in the buffers is refreshed but not refetched, the membership in the rowset is fixed. This is different from the refresh performed by a call to <code>SQLFetchScroll</code> with a FetchOrientation of SQL_FETCH_RELATIVE and a <em>RowNumber</em> equal to 0, which refetches the rowset from the result set so that it can show added data and remove deleted data if those operations are supported by the driver and the cursor. A successful refresh with <code>SQLSetPos</code> will not change a row status of SQL_ROW_DELETED. Deleted rows within the rowset will continue to be marked as deleted until the next fetch. The rows will disappear at the next fetch if the cursor supports packing (in which a subsequent <code>SQLFetch</code> or <code>SQLFetchScroll</code> does not return deleted rows). Added rows do not appear when a refresh with <code>SQLSetPos</code> is performed. This behavior is different from <code>SQLFetchScroll</code> with a FetchType of SQL_FETCH_RELATIVE and a <em>RowNumber</em> equal to 0, which also refreshes the current rowset but will show added records or pack deleted records if these operations are supported by the cursor. A successful refresh with <code>SQLSetPos</code> will change a row status of SQL_ROW_ADDED to SQL_ROW_SUCCESS (if the row status array exists). A successful refresh with <code>SQLSetPos</code> will change a row status of SQL_ROW_UPDATED to the row’s new status (if the row status array exists).</td>
</tr>
</tbody>
</table>
If an error occurs in a SQLSetPos operation on a row, the row status is set to SQL_ROW_ERROR (if the row status array exists).

For a cursor opened with an SQL_ATTR_CONCURRENCY statement attribute of SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES, a refresh with SQLSetPos might update the optimistic concurrency values used by the data source to detect that the row has changed. If this occurs, the row versions or values used to ensure cursor concurrency are updated whenever the rowset buffers are refreshed from the server. This occurs for each row that is refreshed.

The contents of the row status array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute are ignored for the SQL_REFRESH Operation.

SQL_UPDATE

The driver positions the cursor on the row specified by RowNumber and updates the underlying row of data with the values in the rowset buffers (the TargetValuePtr argument in SQLBindCol). It retrieves the lengths of the data from the length/indicator buffers (the StrLen_or_IndPtr argument in SQLBindCol). If the length of any column is SQL_COLUMN_IGNORE, the column is not updated. After updating the row, the driver changes the corresponding element of the row status array to SQL_ROW_UPDATED or SQL_ROW_SUCCESS_WITH_INFO (if the row status array exists).

It is driver-defined what the behavior is if SQLSetPos with an Operation argument of SQL_UPDATE is called on a cursor that contains duplicate columns. The driver can return a driver-defined SQLSTATE, can update the first column that appears in the result set, or perform other driver-defined behavior.

The row operation array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute can be used to indicate that a row in the current rowset should be ignored during a bulk update. For more information, see "Status and Operation Arrays" later in this function reference.

SQL_DELETE

The driver positions the cursor on the row specified by RowNumber and deletes the underlying row of data. It changes the corresponding element of the row status array to SQL_ROW_DELETED. After the row has been deleted, the following are not valid for the row: positioned update and delete statements, calls to SQLGetData, and calls to SQLSetPos with Operation set to anything except SQL_POSITION. For drivers that support packing, the row is deleted from the cursor when new data is retrieved from the data source.

Whether the row remains visible depends on the cursor type. For example, deleted rows are visible to static and keyset-driven cursors but invisible to dynamic cursors.

The row operation array pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute can be used to indicate that a row in the current rowset should be ignored during a bulk delete. For more information, see "Status and Operation Arrays" later in this function reference.

LockType Argument

The LockType argument provides a way for applications to control concurrency. In most cases, data
sources that support concurrency levels and transactions will support only the SQL_LOCK_NO_CHANGE value of the LockType argument. The LockType argument is generally used only for file-based support.

The LockType argument specifies the lock state of the row after SQLSetPos has been executed. If the driver is unable to lock the row either to perform the requested operation or to satisfy the LockType argument, it returns SQL_ERROR and SQLSTATE 42000 (Syntax error or access violation).

Although the LockType argument is specified for a single statement, the lock accords the same privileges to all statements on the connection. In particular, a lock that is acquired by one statement on a connection can be unlocked by a different statement on the same connection.

A row locked through SQLSetPos remains locked until the application calls SQLSetPos for the row with LockType set to SQL_LOCK_UNLOCK, or until the application calls SQLFreeHandle for the statement or SQLFreeStmt with the SQL_CLOSE option. For a driver that supports transactions, a row locked through SQLSetPos is unlocked when the application calls SQLEndTran to commit or roll back a transaction on the connection (if a cursor is closed when a transaction is committed or rolled back, as indicated by the SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR information types returned by SQLGetInfo).

The LockType argument supports the following types of locks. To determine which locks are supported by a data source, an application calls SQLGetInfo with the SQL_DYNAMIC_CURSOR_ATTRIBUTES1, SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1, SQL_KEYSET_CURSOR_ATTRIBUTES1, or SQL_STATIC_CURSOR_ATTRIBUTES1 information type (depending on the type of the cursor).

<table>
<thead>
<tr>
<th>LockType argument</th>
<th>Lock type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_LOCK_NO_CHANGE</td>
<td>The driver or data source ensures that the row is in the same locked or unlocked state as it was before SQLSetPos was called. This value of LockType allows data sources that do not support explicit row-level locking to use whatever locking is required by the current concurrency and transaction isolation levels.</td>
</tr>
<tr>
<td>SQL_LOCK_EXCLUSIVE</td>
<td>The driver or data source locks the row exclusively. A statement on a different connection or in a different application cannot be used to acquire any locks on the row.</td>
</tr>
<tr>
<td>SQL_LOCK_UNLOCK</td>
<td>The driver or data source unlocks the row.</td>
</tr>
</tbody>
</table>

If a driver supports SQL_LOCK_EXCLUSIVE but does not support SQL_LOCK_UNLOCK, a row that is locked will remain locked until one of the function calls described in the previous paragraph occurs.

If a driver supports SQL_LOCK_EXCLUSIVE but does not support SQL_LOCK_UNLOCK, a row that is locked will remain locked until the application calls SQLFreeHandle for the statement or SQLFreeStmt with the SQL_CLOSE option. If the driver supports transactions and closes the cursor upon committing or rolling back the transaction, the application calls SQLEndTran.

For the update and delete operations in SQLSetPos, the application uses the LockType argument as follows:

- To guarantee that a row does not change after it is retrieved, an application calls SQLSetPos with Operation set to SQL_REFRESH and LockType set to SQL_LOCK_EXCLUSIVE.
- If the application sets LockType to SQL_LOCK_NO_CHANGE, the driver guarantees that an
update or delete operation will succeed only if the application specified SQL_CONCUR_LOCK for
the SQL_ATTR_CONCURRENCY statement attribute.

- If the application specifies SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES for the
  SQL_ATTR_CONCURRENCY statement attribute, the driver compares row versions or values and
  rejects the operation if the row has changed since the application fetched the row.

- If the application specifies SQL_CONCUR_READ_ONLY for the SQL_ATTR_CONCURRENCY
  statement attribute, the driver rejects any update or delete operation.

For more information about the SQL_ATTR_CONCURRENCY statement attribute, see SQLSetStmtAttr.

Status and Operation Arrays

The following status and operation arrays are used when calling SQLSetPos:

- The row status array (as pointed to by the SQL_DESC_ARRAY_STATUS_PTR field in the IRD and
  the SQL_ATTR_ROW_STATUS_ARRAY statement attribute) contains status values for each row of
  data in the rowset. The driver sets the status values in this array after a call to SQLFetch,
  SQLFetchScroll, SQLBulkOperations, or SQLSetPos. This array is pointed to by the
  SQL_ATTR_ROW_STATUS_PTR statement attribute.

- The row operation array (as pointed to by the SQL_DESC_ARRAY_STATUS_PTR field in the ARD
  and the SQL_ATTR_ROW_OPERATION_ARRAY statement attribute) contains a value for each row
  in the rowset that indicates whether a call to SQLSetPos for a bulk operation is ignored or
  performed. Each element in the array is set to either SQL_ROW_PROCEED (the default) or
  SQL_ROW_IGNORE. This array is pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement
  attribute.

The number of elements in the status and operation arrays must equal the number of rows in the
rowset (as defined by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute).

For information about the row status array, see SQLFetch. For information about the row operation
array, see "Ignoring a Row in a Bulk Operation," later in this section.

Using SQLSetPos

Before an application calls SQLSetPos, it must perform the following sequence of steps:

1. If the application will call SQLSetPos with Operation set to SQL_UPDATE, call SQLBindCol
   (or SQLSetDescRec) for each column to specify its data type and bind buffers for the column's
data and length.

2. If the application will call SQLSetPos with Operation set to SQL_DELETE or SQL_UPDATE, call
   SQLColAttribute to make sure that the columns to be deleted or updated are updatable.

3. Call SQLExecDirect, SQLExecute, or a catalog function to create a result set.

4. Call SQLFetch or SQLFetchScroll to retrieve the data.
For more information about using `SQLSetPos`, see Updating Data with SQLSetPos.

Deleting Data Using SQLSetPos

To delete data with `SQLSetPos`, an application calls `SQLSetPos` with `RowNumber` set to the number of the row to delete and `Operation` set to SQL_DELETE.

After the data has been deleted, the driver changes the value in the implementation row status array for the appropriate row to SQL_ROW_DELETED (or SQL_ROW_ERROR).

Updating Data Using SQLSetPos

An application can pass the value for a column either in the bound data buffer or with one or more calls to `SQLPutData`. Columns whose data is passed with `SQLPutData` are known as data-at-execution columns. These are commonly used to send data for SQL_LONGVARBINARY and SQL_LONGVARCHAR columns and can be mixed with other columns.

To update data with `SQLSetPos`, an application:

1. Places values in the data and length/indicator buffers bound with `SQLBindCol`:
   - For normal columns, the application places the new column value in the `*TargetValuePtr` buffer and the length of that value in the `*StrLen_or_IndPtr` buffer. If the row should not be updated, the application places SQL_ROW_IGNORE in that row's element of the row operation array.
   - For data-at-execution columns, the application places an application-defined value, such as the column number, in the `*TargetValuePtr` buffer. The value can be used later to identify the column.
   The application places the result of the SQL_LEN_DATA_AT_EXEC(length) macro in the `*StrLen_or_IndPtr` buffer. If the SQL data type of the column is SQL_LONGVARBINARY, SQL_LONGVARCHAR, or a long data source-specific data type and the driver returns “Y” for the SQL_NEED_LONG_DATA_LEN information type in `SQLGetInfo`, `length` is the number of bytes of data to be sent for the parameter; otherwise, it must be a non-negative value and is ignored.

2. Calls `SQLSetPos` with the `Operation` argument set to SQL_UPDATE to update the row of data.
   - If there are no data-at-execution columns, the process is complete.
   - If there are any data-at-execution columns, the function returns SQL_NEED_DATA and proceeds to step 3.

3. Calls `SQLParamData` to retrieve the address of the `*TargetValuePtr` buffer for the first data-at-execution column to be processed. `SQLParamData` returns SQL_NEED_DATA. The application retrieves the application-defined value from the `*TargetValuePtr` buffer.

**Note**

Although data-at-execution parameters are similar to data-at-execution columns, the value returned by `SQLParamData` is different for each.
Data-at-execution parameters are parameters in an SQL statement for which data will be sent with SQLPutData when the statement is executed with SQLExecDirect or SQLExecute. They are bound with SQLBindParameter or by setting descriptors with SQLSetDescRec. The value returned by SQLParamData is a 32-bit value passed to SQLBindParameter in the ParameterValuePtr argument.

Data-at-execution columns are columns in a rowset for which data will be sent with SQLPutData when a row is updated with SQLSetPos. They are bound with SQLBindCol. The value returned by SQLParamData is the address of the row in the *TargetValuePtr buffer that is being processed.

4. Calls SQLPutData one or more times to send data for the column. More than one call is needed if all the data values cannot be returned in the *TargetValuePtr buffer specified in SQLPutData; multiple calls to SQLPutData for the same column are allowed only when sending character C data to a column with a character, binary, or data source–specific data type or when sending binary C data to a column with a character, binary, or data source–specific data type.

5. Calls SQLParamData again to signal that all data has been sent for the column. If there are more data-at-execution columns, SQLParamData returns SQL_NEED_DATA and the address of the TargetValuePtr buffer for the next data-at-execution column to be processed. The application repeats steps 4 and 5.

If there are no more data-at-execution columns, the process is complete. If the statement was executed successfully, SQLParamData returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO; if the execution failed, it returns SQL_ERROR. At this point, SQLParamData can return any SQLSTATE that can be returned by SQLSetPos.

If data has been updated, the driver changes the value in the implementation row status array for the appropriate row to SQL_ROW_UPDATED.

If the operation is canceled or an error occurs in SQLParamData or SQLPutData, after SQLSetPos returns SQL_NEED_DATA and before data is sent for all data-at-execution columns, the application can call only SQLCancel, SQLGetDiagField, SQLGetDiagRec, SQLGetFunctions, SQLParamData, or SQLPutData for the statement or the connection associated with the statement. If it calls any other function for the statement or the connection associated with the statement, the function returns SQL_ERROR and SQLSTATE HY010 (Function sequence error).

If the application calls SQLCancel while the driver still needs data for data-at-execution columns, the driver cancels the operation. The application can then call SQLSetPos again; canceling does not affect the cursor state or the current cursor position.

When the SELECT-list of the query specification associated with the cursor contains more than one reference to the same column, whether an error is generated or the driver ignores the duplicated references and performs the requested operations is driver-defined.

Performing Bulk Operations

If the RowNumber argument is 0, the driver performs the operation specified in the Operation
argument for every row in the rowset that has a value of SQL_ROW_PROCEED in its field in the row operation array pointed to by SQL_ATTR_ROW_OPERATION_PTR statement attribute. This is a valid value of the RowNumber argument for an Operation argument of SQL_DELETE, SQL_REFRESH, or SQL_UPDATE, but not SQL_POSITION. SQLSetPos with an Operation of SQL_POSITION and a RowNumber equal to 0 will return SQLSTATE HY109 (Invalid cursor position).

If an error occurs that pertains to the entire rowset, such as SQLSTATE HYT00 (Timeout expired), the driver returns SQL_ERROR and the appropriate SQLSTATE. The contents of the rowset buffers are undefined, and the cursor position is unchanged.

If an error occurs that pertains to a single row, the driver:

- Sets the element for the row in the row status array pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute to SQL_ROW_ERROR.
- Posts one or more additional SQLSTATEs for the error in the error queue and sets the SQL_DIAG_ROW_NUMBER field in the diagnostic data structure.

After it has processed the error or warning, if the driver completes the operation for the remaining rows in the rowset, it returns SQL_SUCCESS_WITH_INFO. Thus, for each row that returned an error, the error queue contains zero or more additional SQLSTATEs. If the driver stops the operation after it has processed the error or warning, it returns SQL_ERROR.

If the driver returns any warnings, such as SQLSTATE 01004 (Data truncated), it returns warnings that apply to the entire rowset or to unknown rows in the rowset before it returns the error information that applies to specific rows. It returns warnings for specific rows along with any other error information about those rows.

If RowNumber is equal to 0 and Operation is SQL_UPDATE, SQL_REFRESH, or SQL_DELETE, the number of rows that SQLSetPos operates on is pointed to by the SQL_ATTR_ROWS_FETCHED_PTR statement attribute.

If RowNumber is equal to 0 and Operation is SQL_DELETE, SQL_REFRESH, or SQL_UPDATE, the current row after the operation is the same as the current row before the operation.

**Ignoring a Row in a Bulk Operation**

The row operation array can be used to indicate that a row in the current rowset should be ignored during a bulk operation using SQLSetPos. To direct the driver to ignore one or more rows during a bulk operation, an application should perform the following steps:

1. Call SQLSetStmtAttr to set the SQL_ATTR_ROW_OPERATION_PTR statement attribute to point to an array of SQLUSMALLINTs. This field can also be set by calling SQLSetDescField to set the SQL_DESC_ARRAY_STATUS_PTR header field of the ARD, which requires that an application obtains the descriptor handle.

2. Set each element of the row operation array to one of two values:
   - SQL_ROW_IGNORE, to indicate that the row is excluded for the bulk operation.
   - SQL_ROW_PROCEED, to indicate that the row is included in the bulk operation. (This is the default value.)

3. Call SQLSetPos to perform the bulk operation.
The following rules apply to the row operation array:

- SQL_ROW_IGNORE and SQL_ROW_PROCEED affect only bulk operations using `SQLSetPos` with an Operation of SQL_DELETE or SQL_UPDATE. They do not affect calls to `SQLSetPos` with an Operation of SQL_REFRESH or SQL_POSITION.

- The pointer is set to null by default.

- If the pointer is null, all rows are updated as if all elements were set to SQL_ROW_PROCEED.

- Setting an element to SQL_ROW_PROCEED does not guarantee that the operation will occur on that particular row. For example, if a certain row in the rowset has the status SQL_ROW_ERROR, the driver might not be able to update that row regardless of whether the application specified SQL_ROW_PROCEED. An application must always check the row status array to see whether the operation was successful.

- SQL_ROW_PROCEED is defined as 0 in the header file. An application can initialize the row operation array to 0 in order to process all rows.

- If element number "n" in the row operation array is set to SQL_ROW_IGNORE and `SQLSetPos` is called to perform a bulk update or delete operation, the nth row in the rowset remains unchanged after the call to `SQLSetPos`.

- An application should automatically set a read-only column to SQL_ROW_IGNORE.

**Ignoring a Column in a Bulk Operation**

To avoid unnecessary processing diagnostics generated by attempted updates to one or more read-only columns, an application can set the value in the bound length/indicator buffer to SQL_COLUMN_IGNORE. For more information, see `SQLBindCol`.

**Code Example**

In the following example, an application allows a user to browse the ORDERS table and update order status. The cursor is keyset-driven with a rowset size of 20 and uses optimistic concurrency control comparing row versions. After each rowset is fetched, the application prints it and allows the user to select and update the status of an order. The application uses `SQLSetPos` to position the cursor on the selected row and performs a positioned update of the row. (Error handling is omitted for clarity.)

```sql
#define ROWS 20
#define STATUS_LEN 6

SQLCHAR szStatus[ROWS][STATUS_LEN], szReply[3];
SQLINTEGER cbStatus[ROWS], cbOrderID;
SQLUSMALLINT rgfRowStatus[ROWS];
SQLUINTEGER sOrderID, crow = ROWS, irow;
SQLHSTMT hstmtS, hstmtU;
```
For more examples, see Positioned Update and Delete Statements and Updating Rows in the Rowset with SQLSetPos.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Performing bulk operations that do not relate to the block cursor position</td>
<td>SQLBulkOperations Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Getting a single field of a descriptor</td>
<td>SQLGetDescField Function</td>
</tr>
</tbody>
</table>
### SQLSetScrollOptions Function

**Conformance**

Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

**Summary**

In ODBC 3.x, the ODBC 2.0 function `SQLSetScrollOptions` has been replaced by calls to `SQLGetInfo` and `SQLSetStmtAttr`.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>When the Driver Manager maps <code>SQLSetScrollOptions</code> for an application working with an ODBC 3.x driver that does not support <code>SQLSetScrollOptions</code>, the Driver Manager sets the SQL_ROWSET_SIZE statement option, not the SQL_ATTR_ROW_ARRAY_SIZE statement attribute, to the RowsetSize argument in <code>SQLSetScrollOption</code>. As a result, <code>SQLSetScrollOptions</code> cannot be used by an application when fetching multiple rows by a call to <code>SQLFetch</code> or <code>SQLFetchScroll</code>. It can be used only when fetching multiple rows by a call to <code>SQLExtendedFetch</code>.</td>
</tr>
</tbody>
</table>

**Remarks**

If your application will run on a 64-bit operating system, see ODBC 64-Bit Information.
SQLSetStmtAttr Function

Conformance
Version Introduced: ODBC 3.0 Standards Compliance: ISO 92

Summary
SQLSetStmtAttr sets attributes related to a statement.

Note
For more information about what the Driver Manager maps this function to when an ODBC 3.x application is working with an ODBC 2.x driver, see Mapping Replacement Functions for Backward Compatibility of Applications.

Syntax

```
SQLRETURN SQLSetStmtAttr(
    SQLHSTMT StatementHandle,
    SQLINTEGER Attribute,
    SQLPOINTER ValuePtr,
    SQLINTEGER StringLength);
```

Arguments

- **StatementHandle**
  [Input] Statement handle.

- **Attribute**
  [Input] Option to set, listed in "Comments."

- **ValuePtr**
  [Input] Value to be associated with **Attribute**. Depending on the value of **Attribute**, **ValuePtr** will be one of the following:
  
  - An ODBC descriptor handle.
  - A SQLUINTEGER value.
A SQLULEN value.

- A pointer to one of the following:
  - A null-terminated character string.
  - A binary buffer.
  - A value or array of type SQLLEN, SQLULEN, or SQLUSMALLINT.
  - A driver-defined value.

If the `Attribute` argument is a driver-specific value, `ValuePtr` may be a signed integer.

### StringLength

[Input] If `Attribute` is an ODBC-defined attribute and `ValuePtr` points to a character string or a binary buffer, this argument should be the length of `*ValuePtr`. If `Attribute` is an ODBC-defined attribute and `ValuePtr` is an integer, `StringLength` is ignored.

If `Attribute` is a driver-defined attribute, the application indicates the nature of the attribute to the Driver Manager by setting the `StringLength` argument. `StringLength` can have the following values:

- If `ValuePtr` is a pointer to a character string, then `StringLength` is the length of the string or SQL_NT.
- If `ValuePtr` is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(`length`) macro in `StringLength`. This places a negative value in `StringLength`.
- If `ValuePtr` is a pointer to a value other than a character string or a binary string, then `StringLength` should have the value SQL_IS_POINTER.
- If `ValuePtr` contains a fixed-length value, then `StringLength` is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.

### Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

### Diagnostics

When `SQLSetStmtAttr` returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling `SQLGetDiagRec` with a `HandleType` of SQL_HANDLE_STMT and a `Handle` of `StatementHandle`. The following table lists the SQLSTATE values commonly returned by `SQLSetStmtAttr` and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed</td>
<td>The driver did not support the value specified in ValuePtr, or the value specified in ValuePtr was invalid because of implementation working conditions, so the driver substituted a similar value. <em>(SQLGetStmtAttr can be called to determine the temporarily substituted value.)</em> The substitute value is valid for the StatementHandle until the cursor is closed, at which point the statement attribute reverts to its previous value. The statement attributes that can be changed are: <strong>SQL_ATTR_CONCURRENCY SQL_ATTR_CURSOR_TYPE SQL_ATTR_KEYSET_SIZE SQL_ATTR_MAX_LENGTH SQL_ATTR_MAX_ROWS SQL_ATTR_QUERY_TIMEOUT SQL_ATTR_ROW_ARRAY_SIZE SQL_ATTR_SIMULATE_CURSOR</strong> <em>(Function returns SQL_SUCCESS_WITH_INFO.)</em></td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>The Attribute was SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, SQL_ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS, and the cursor was open.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the <strong>MessageText</strong> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>The Attribute argument identified a statement attribute that required a string attribute, and the ValuePtr argument was a null pointer.</td>
</tr>
</tbody>
</table>
| HY010      | Function sequence error | *(DM)* An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLSetStmtAttr function was called. *(DM)* **SQLExecute, SQLExecDirect, or SQLMoreResults** was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. *(DM)* An asynchronously executing function was called for the StatementHandle and was still executing when this function was called. *(DM)* **SQLExecute, SQLExecDirect, SQLBulkOperations, or**
**SQLSetPos** was called for the **StatementHandle** and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY011</td>
<td>Attribute cannot be set now</td>
<td>The <strong>Attribute</strong> was SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, SQL_ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS, and the statement was prepared.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
</tbody>
</table>
| HY017 | Invalid use of an automatically allocated descriptor handle                | (DM) The **Attribute** argument was SQL_ATTR_IMP_ROW_DESC or SQL_ATTR_IMP_PARAM_DESC.  
   (DM) The **Attribute** argument was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and the value in **ValuePtr** was an implicitly allocated descriptor handle other than the handle originally allocated for the ARD or APD. |
| HY024 | Invalid attribute value                                                    | Given the specified **Attribute** value, an invalid value was specified in **ValuePtr**. (The Driver Manager returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE or SQL_ATTR_ASYNC_ENABLE. For all other connection and statement attributes, the driver must verify the value specified in **ValuePtr**.)  
   The **Attribute** argument was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and **ValuePtr** was an explicitly allocated descriptor handle that is not on the same connection as the **StatementHandle** argument. |
| HY090 | Invalid string or buffer length                                            | (DM) **ValuePtr** is a character string, and the **StringLength** argument was less than 0 but was not SQL_NTS. |
| HY092 | Invalid attribute/option identifier                                       | (DM) The value specified for the argument **Attribute** was not valid for the version of ODBC supported by the driver.  
   (DM) The value specified for the argument **Attribute** was a read-only attribute. |
| HY117 | Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed. | (DM) For more information about suspended state, see [SQLSetPos](#) Function. |
| HYC00 | Optional feature not implemented                                           | The value specified for the argument **Attribute** was a valid ODBC statement attribute for the version of ODBC supported by the driver but was not supported by the driver. |
The Attribute argument was SQL_ATTR_ASYNC_ENABLE, and a call to SQLGetInfo with an InfoType of SQL_ASYNC_MODE returns SQL_AM_CONNECTION.

The Attribute argument was SQL_ATTR_ENABLE_AUTO_IPD, and the value of the connection attribute SQL_ATTR_AUTO_IPD was SQL_FALSE.

<table>
<thead>
<tr>
<th>HYT01</th>
<th>Connection timeout expired</th>
<th>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
<tr>
<td>S1118</td>
<td>Driver does not support asynchronous notification</td>
<td>If calling SQLSetStmtAttr to set SQL_ATTR_ASYNC_STMT_EVENT; asynchronous notification is not supported by the driver.</td>
</tr>
</tbody>
</table>

Comments

Statement attributes for a statement remain in effect until they are changed by another call to SQLSetStmtAttr or until the statement is dropped by calling SQLFreeHandle. Calling SQLFreeStmt with the SQL_CLOSE, SQL_UNBIND, or SQL_RESET_PARAMS option does not reset statement attributes.

Some statement attributes support substitution of a similar value if the data source does not support the value specified in ValuePtr. In such cases, the driver returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed). For example, if Attribute is SQL_ATTR_CONCURRENCY and ValuePtr is SQL_CONCUR_ROWVER, and if the data source does not support this, the driver substitutes SQL_CONCUR_VALUES and returns SQL_SUCCESS_WITH_INFO. To determine the substituted value, an application calls SQLGetStmtAttr.

The format of information set with ValuePtr depends on the specified Attribute. SQLSetStmtAttr accepts attribute information in one of two different formats: a character string or an integer value. The format of each is noted in the attribute's description. This format applies to the information returned for each attribute in SQLGetStmtAttr. Character strings pointed to by the ValuePtr argument of SQLSetStmtAttr have a length of StringLength.

Note

The ability to set statement attributes at the connection level by calling SQLSetConnectAttr has been deprecated in ODBC 3.x. ODBC 3.x applications should never set statement attributes at the connection level. ODBC 3.x statement attributes cannot be set at the connection level, with the exception of the SQL_ATTR_METADATA_ID and SQL_ATTR_ASYNC_ENABLE attributes, which are both connection attributes and statement attributes, and can be set at either the connection level or the statement level.

Note
ODBC 3.x drivers need only support this functionality if they should work with ODBC 2.x applications that set ODBC 2.x statement options at the connection level. For more information, see "Setting Statement Options on the Connection Level" under SQLSetConnectOption Mapping in Appendix G: Driver Guidelines for Backward Compatibility.

Statement Attributes That Set Descriptor Fields

Many statement attributes correspond to a header field of a descriptor. Setting these attributes actually results in the setting of the descriptor fields. Setting fields by a call to SQLSetStmtAttr rather than to SQLSetDescField has the advantage that a descriptor handle does not have to be obtained for the function call.

**Caution**

Calling SQLSetStmtAttr for one statement can affect other statements. This occurs when the APD or ARD associated with the statement is explicitly allocated and is also associated with other statements. Because SQLSetStmtAttr modifies the APD or ARD, the modifications apply to all statements with which this descriptor is associated. If this is not the required behavior, the application should dissociate this descriptor from the other statements (by calling SQLSetStmtAttr to set the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC field to a different descriptor handle) before calling SQLSetStmtAttr again.

When a descriptor field is set as a result of the corresponding statement attribute being set, the field is set only for the applicable descriptors that are currently associated with the statement identified by the StatementHandle argument, and the attribute setting does not affect any descriptors that may be associated with that statement in the future. When a descriptor field that is also a statement attribute is set by a call to SQLSetDescField, the corresponding statement attribute is set. If an explicitly allocated descriptor is dissociated from a statement, a statement attribute that corresponds to a header field will revert to the value of the field in the implicitly allocated descriptor.

When a statement is allocated (see SQLAllocHandle), four descriptor handles are automatically allocated and associated with the statement. Explicitly allocated descriptor handles can be associated with the statement by calling SQLAllocHandle with an HandleType of SQL_HANDLE_DESC to allocate a descriptor handle and then calling SQLSetStmtAttr to associate the descriptor handle with the statement.

The statement attributes in the following table correspond to descriptor header fields.

<table>
<thead>
<tr>
<th>Statement attribute</th>
<th>Header field</th>
<th>Desc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_PARAM_BIND_OFFSET_PTR</td>
<td>SQL_DESC_BIND_OFFSET_PTR</td>
<td>APD</td>
</tr>
<tr>
<td>SQL_ATTR_PARAM_BIND_TYPE</td>
<td>SQL_DESC_BIND_TYPE</td>
<td>APD</td>
</tr>
<tr>
<td>SQL_ATTR_PARAM_OPERATION_PTR</td>
<td>SQL_DESC_ARRAY_STATUS_PTR</td>
<td>APD</td>
</tr>
<tr>
<td>SQL_ATTR_PARAM_STATUS_PTR</td>
<td>SQL_DESC_ARRAY_STATUS_PTR</td>
<td>IPD</td>
</tr>
<tr>
<td>SQL_ATTR_PARAMS_PROCESSED_PTR</td>
<td>SQL_DESC_ROWS_PROCESSED_PTR</td>
<td>IPD</td>
</tr>
<tr>
<td>SQL_ATTR_PARAMSET_SIZE</td>
<td>SQL_DESC_ARRAY_SIZE</td>
<td>APD</td>
</tr>
</tbody>
</table>
Statement Attributes

The currently defined attributes and the version of ODBC in which they were introduced are shown in the following table; it is expected that more attributes will be defined by drivers to take advantage of different data sources. A range of attributes is reserved by ODBC; driver developers must reserve values for their own driver-specific use from Open Group. For more information, see Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>ValuePtr contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_APP_PARAM_DESC (ODBC 3.0)</td>
<td>The handle to the APD for subsequent calls to SQLExecute and SQLExecDirect on the statement handle. The initial value of this attribute is the descriptor implicitly allocated when the statement was initially allocated. If the value of this attribute is set to SQL_NULL_DESC or the handle originally allocated for the descriptor, an explicitly allocated APD handle that was previously associated with the statement handle is dissociated from it and the statement handle reverts to the implicitly allocated APD handle. This attribute cannot be set to a descriptor handle that was implicitly allocated for another statement or to another descriptor handle that was implicitly set on the same statement; implicitly allocated descriptor handles cannot be associated with more than one statement or descriptor handle.</td>
</tr>
<tr>
<td>SQL_ATTR_APP_ROW_DESC (ODBC 3.0)</td>
<td>The handle to the ARD for subsequent fetches on the statement handle. The initial value of this attribute is the descriptor implicitly allocated when the statement was initially allocated. If the value of this attribute is set to SQL_NULL_DESC or the handle originally allocated for the descriptor, an explicitly allocated ARD handle that was previously associated with the statement handle is dissociated from it and the statement handle reverts to the implicitly allocated ARD handle.</td>
</tr>
</tbody>
</table>
This attribute cannot be set to a descriptor handle that was implicitly allocated for another statement or to another descriptor handle that was implicitly set on the same statement; implicitly allocated descriptor handles cannot be associated with more than one statement or descriptor handle.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_ASYNC_ENABLE (ODBC 1.0)</td>
<td>A SQLULEN value that specifies whether a function called with the specified statement is executed asynchronously:</td>
</tr>
<tr>
<td></td>
<td>SQL_ASYNC_ENABLE_OFF = Disable statement level asynchronous execution support (the default).</td>
</tr>
<tr>
<td></td>
<td>SQL_ASYNC_ENABLE_ON = Enable statement level asynchronous execution support.</td>
</tr>
<tr>
<td></td>
<td>For more information, see Asynchronous Execution (Polling Method).</td>
</tr>
<tr>
<td></td>
<td>For drivers with statement level asynchronous execution support, the statement attribute SQL_ATTR_ASYNC_ENABLE is read only. Its value is the same as the value of the connection level attribute with the same name at the time the statement handle was allocated.</td>
</tr>
<tr>
<td></td>
<td>Calling SQLSetStmtAttr to set SQL_ATTR_ASYNC_ENABLE when the SQL_ASYNC_MODE InfoType returns SQL_AM_CONNECTION returns SQLSTATE HYC00 (Optional feature not implemented). For more information, see SQLSetConnectAttr Function for more information.</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_STMT_EVENT (ODBC 3.8)</td>
<td>A SQLPOINTER value that is an event handle.</td>
</tr>
<tr>
<td></td>
<td>Notification of completion of asynchronous functions is enabled by calling SQLSetStmtAttr to set the SQL_ATTR_ASYNC_STMT_EVENT attribute and specify the event handle.</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_STMT_PCALLBACK (ODBC 3.8)</td>
<td>A SQLPOINTER to the asynchronous callback function.</td>
</tr>
<tr>
<td></td>
<td>Only the Driver Manager can call a driver’s SQLSetStmtAttr function with this attribute.</td>
</tr>
<tr>
<td>SQL_ATTR_ASYNC_STMT_PCONTEXT (ODBC 3.8)</td>
<td>A SQLPOINTER to the context structure</td>
</tr>
<tr>
<td></td>
<td>Only the Driver Manager can call a driver’s SQLSetStmtAttr function with this attribute.</td>
</tr>
<tr>
<td>SQL_ATTR_CONCURRENCY (ODBC 2.0)</td>
<td>An SQLULEN value that specifies the cursor concurrency:</td>
</tr>
<tr>
<td></td>
<td>SQL_CONCUR_READ_ONLY = Cursor is read-only. No updates are allowed.</td>
</tr>
<tr>
<td></td>
<td>SQL_CONCUR_LOCK = Cursor uses the lowest level of locking sufficient to ensure that the row can be updated.</td>
</tr>
</tbody>
</table>
SQL_CONCUR_ROWVER = Cursor uses optimistic concurrency control, comparing row versions such as SQLBase ROWID or Sybase TIMESTAMP.

SQL_CONCUR_VALUES = Cursor uses optimistic concurrency control, comparing values.

The default value for SQL_ATTR_CONCURRENCY is SQL_CONCUR_READ_ONLY.

This attribute cannot be specified for an open cursor. For more information, see Concurrency Types.

If the SQL_ATTR_CURSOR_TYPE Attribute is changed to a type that does not support the current value of SQL_ATTR_CONCURRENCY, the value of SQL_ATTR_CONCURRENCY will be changed at execution time, and a warning issued when SQLExecDirect or SQLPrepare is called.

If the driver supports the SELECT FOR UPDATE statement and such a statement is executed while the value of SQL_ATTR_CONCURRENCY is set to SQL_CONCUR_READ_ONLY, an error will be returned. If the value of SQL_ATTR_CONCURRENCY is changed to a value that the driver supports for some value of SQL_ATTR_CURSOR_TYPE but not for the current value of SQL_ATTR_CURSOR_TYPE, the value of SQL_ATTR_CURSOR_TYPE will be changed at execution time and SQLSTATE 01S02 (Option value changed) is issued when SQLExecDirect or SQLPrepare is called.

If the specified concurrency is not supported by the data source, the driver substitutes a different concurrency and returns SQLSTATE 01S02 (Option value changed). For SQL_CONCUR_VALUES, the driver substitutes SQL_CONCUR_ROWVER, and vice versa. For SQL_CONCUR_LOCK, the driver substitutes, in order, SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES. The validity of the substituted value is not checked until execution time.

For more information about the relationship between SQL_ATTR_CONCURRENCY and the other cursor attributes, see Cursor Characteristics and Cursor Type.

<table>
<thead>
<tr>
<th>SQL_ATTR_CURSOR_SCROLLABLE (ODBC 3.0)</th>
<th>An SQLULEN value that specifies the level of support that the application requires. Setting this attribute affects subsequent calls to SQLExecDirect and SQLExecute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_NONSCROLLABLE = Scrollable cursors are not required on the statement handle. If the application calls SQLFetchScroll on this handle, the only valid value of FetchOrientation is SQL_FETCH_NEXT. This is the default.</td>
<td>SQL_SCROLLABLE = Scrollable cursors are required on</td>
</tr>
</tbody>
</table>
the statement handle. When calling `SQLFetchScroll`, the application may specify any valid value of `FetchOrientation`, achieving cursor positioning in modes other than the sequential mode.

For more information about scrollable cursors, see [Scrollable Cursors](#). For more information about the relationship between `SQL_ATTR_CURSOR_SCROLLABLE` and the other cursor attributes, see [Cursor Characteristics and Cursor Type](#).

| **SQL_ATTR_CURSOR_SENSITIVITY (ODBC 3.0)** | An SQLULEN value that specifies whether cursors on the statement handle make visible the changes made to a result set by another cursor. Setting this attribute affects subsequent calls to `SQLExecDirect` and `SQLExecute`. An application can read back the value of this attribute to obtain its initial state or its state as most recently set by the application.

- **SQL_UNSPECIFIED** = It is unspecified what the cursor type is and whether cursors on the statement handle make visible the changes made to a result set by another cursor. Cursors on the statement handle may make visible none, some, or all such changes. This is the default.

- **SQL_INSENSITIVE** = All cursors on the statement handle show the result set without reflecting any changes made to it by any other cursor. Inensitive cursors are read-only. This corresponds to a static cursor, which has a concurrency that is read-only.

- **SQL_SENSITIVE** = All cursors on the statement handle make visible all changes made to a result set by another cursor.

For more information about the relationship between `SQL_ATTR_CURSOR_SENSITIVITY` and the other cursor attributes, see [Cursor Characteristics and Cursor Type](#).

| **SQL_ATTR_CURSOR_TYPE (ODBC 2.0)** | An SQLULEN value that specifies the cursor type:

- **SQL_CURSOR_FORWARD_ONLY** = The cursor only scrolls forward.

- **SQL_CURSOR_STATIC** = The data in the result set is static.

- **SQL_CURSOR_KEYSET_DRIVEN** = The driver saves and uses the keys for the number of rows specified in the `SQL_ATTR_KEYSET_SIZE` statement attribute.

- **SQL_CURSOR_DYNAMIC** = The driver saves and uses only the keys for the rows in the rowset.

The default value is `SQL_CURSOR_FORWARD_ONLY`. This attribute cannot be specified after the SQL statement has been prepared.
If the specified cursor type is not supported by the data source, the driver substitutes a different cursor type and returns SQLSTATE 01S02 (Option value changed). For a mixed or dynamic cursor, the driver substitutes, in order, a keyset-driven or static cursor. For a keyset-driven cursor, the driver substitutes a static cursor.

For more information about scrollable cursor types, see Scrollable Cursor Types. For more information about the relationship between SQL_ATTR_CURSOR_TYPE and the other cursor attributes, see Cursor Characteristics and Cursor Type.

| SQL_ATTR_ENABLE_AUTO_IPD (ODBC 3.0) | An SQLULEN value that specifies whether automatic population of the IPD is performed:
| SQL_TRUE = Turns on automatic population of the IPD after a call to SQLPrepare. SQL_FALSE = Turns off automatic population of the IPD after a call to SQLPrepare. (An application can still obtain IPD field information by calling SQLDescribeParam, if supported.) The default value of the statement attribute SQL_ATTR_ENABLE_AUTO_IPD is SQL_FALSE. For more information, see Automatic Population of the IPD. |
| SQL_ATTR_FETCH_BOOKMARK_PTR (ODBC 3.0) | A SQLLEN * that points to a binary bookmark value. When SQLFetchScroll is called with fFetchOrientation equal to SQL_FETCH_BOOKMARK, the driver picks up the bookmark value from this field. This field defaults to a null pointer. For more information, see Scrolling by Bookmark.
| The value pointed to by this field is not used for delete by bookmark, update by bookmark, or fetch by bookmark operations in SQLBulkOperations, which use bookmarks cached in rowset buffers. |
| SQL_ATTR_IMP_PARAM_DESC (ODBC 3.0) | The handle to the IPD. The value of this attribute is the descriptor allocated when the statement was initially allocated. The application cannot set this attribute. This attribute can be retrieved by a call to SQLGetStmtAttr but not set by a call to SQLSetStmtAttr. |
| SQL_ATTR_IMP_ROW_DESC (ODBC 3.0) | The handle to the IRD. The value of this attribute is the descriptor allocated when the statement was initially allocated. The application cannot set this attribute. This attribute can be retrieved by a call to SQLGetStmtAttr but not set by a call to SQLSetStmtAttr. |
| SQL_ATTR_KEYSET_SIZE (ODBC 2.0) | An SQLULEN that specifies the number of rows in the keyset for a keyset-driven cursor. If the keyset size is 0 (the default), the cursor is fully keyset-driven. If the... |
keyset size is greater than 0, the cursor is mixed (keyset-driven within the keyset and dynamic outside of the keyset). The default keyset size is 0. For more information about keyset-driven cursors, see Keyset-Driven Cursors.

If the specified size exceeds the maximum keyset size, the driver substitutes that size and returns SQLSTATE 01S02 (Option value changed).

SQLFetch or SQLFetchScroll returns an error if the keyset size is greater than 0 and less than the rowset size.

SQL_ATTR_MAX_LENGTH (ODBC 1.0)

An SQLULEN value that specifies the maximum amount of data that the driver returns from a character or binary column. If ValuePtr is less than the length of the available data, SQLFetch or SQLGetData truncates the data and returns SQL_SUCCESS. If ValuePtr is 0 (the default), the driver attempts to return all available data.

If the specified length is less than the minimum amount of data that the data source can return or greater than the maximum amount of data that the data source can return, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed).

The value of this attribute can be set on an open cursor; however, the setting might not take effect immediately, in which case the driver will return SQLSTATE 01S02 (Option value changed) and reset the attribute to its original value.

This attribute is intended to reduce network traffic and should be supported only when the data source (as opposed to the driver) in a multiple-tier driver can implement it. This mechanism should not be used by applications to truncate data; to truncate data received, an application should specify the maximum buffer length in the BufferLength argument in SQLBindCol or SQLGetData.

SQL_ATTR_MAX_ROWS (ODBC 1.0)

An SQLULEN value corresponding to the maximum number of rows to return to the application for a SELECT statement. If *ValuePtr equals 0 (the default), the driver returns all rows.

This attribute is intended to reduce network traffic. Conceptually, it is applied when the result set is created and limits the result set to the first ValuePtr rows. If the number of rows in the result set is greater than ValuePtr, the result set is truncated.

SQL_ATTR_MAX_ROWS applies to all result sets on the Statement, including those returned by catalog functions. SQL_ATTR_MAX_ROWS establishes a maximum for the value of the cursor row count.
A driver should not emulate SQL_ATTR_MAX_ROWS behavior for SQLFetch or SQLFetchScroll (if result set size limitations cannot be implemented at the data source) if it cannot guarantee that SQL_ATTR_MAX_ROWS will be implemented properly.

It is driver-defined whether SQL_ATTR_MAX_ROWS applies to statements other than SELECT statements (such as catalog functions).

The value of this attribute can be set on an open cursor; however, the setting might not take effect immediately, in which case the driver will return SQLSTATE 01S02 (Option value changed) and reset the attribute to its original value.

SQL_ATTR_METADATA_ID (ODBC 3.0)

An SQLULEN value that determines how the string arguments of catalog functions are treated.

If SQL_TRUE, the string argument of catalog functions are treated as identifiers. The case is not significant. For nondelimited strings, the driver removes any trailing spaces and the string is folded to uppercase. For delimited strings, the driver removes any leading or trailing spaces and takes whatever is between the delimiters literally. If one of these arguments is set to a null pointer, the function returns SQL_ERROR and SQLSTATE HY009 (Invalid use of null pointer).

If SQL_FALSE, the string arguments of catalog functions are not treated as identifiers. The case is significant. They can either contain a string search pattern or not, depending on the argument.

The default value is SQL_FALSE.

The TableType argument of SQLTables, which takes a list of values, is not affected by this attribute.

SQL_ATTR_METADATA_ID can also be set on the connection level. (It and SQL_ATTR_ASYNC_ENABLE are the only statement attributes that are also connection attributes.)

For more information, see Arguments in Catalog Functions.

SQL_ATTR_NOSCAN (ODBC 1.0)

An SQLULEN value that indicates whether the driver should scan SQL strings for escape sequences:

SQL_NOSCAN_OFF = The driver scans SQL strings for escape sequences (the default).

SQL_NOSCAN_ON = The driver does not scan SQL strings for escape sequences. Instead, the driver sends the statement directly to the data source.
**SQL_ATTR_PARAM_BIND_OFFSET_PTR (ODBC 3.0)**

An SQLULEN * value that points to an offset added to pointers to change binding of dynamic parameters. If this field is non-null, the driver dereferences the pointer, adds the dereferenced value to each of the deferred fields in the descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR), and uses the new pointer values when binding. It is set to null by default.

The bind offset is always added directly to the SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR fields. If the offset is changed to a different value, the new value is still added directly to the value in the descriptor field. The new offset is not added to the field value plus any earlier offsets.

For more information, see [Parameter Binding Offsets](#).

Setting this statement attribute sets the SQL_DESC_BIND_OFFSET_PTR field in the APD header.

---

**SQL_ATTR_PARAM_BIND_TYPE (ODBC 3.0)**

An SQLULEN value that indicates the binding orientation to be used for dynamic parameters.

This field is set to SQL_PARAM_BIND_BY_COLUMN (the default) to select column-wise binding.

To select row-wise binding, this field is set to the length of the structure or an instance of a buffer that will be bound to a set of dynamic parameters. This length must include space for all of the bound parameters and any padding of the structure or buffer to ensure that when the address of a bound parameter is incremented with the specified length, the result will point to the beginning of the same parameter in the next set of parameters. When using the `sizeof` operator in ANSI C, this behavior is guaranteed.

For more information, see [Binding Arrays of Parameters](#).

Setting this statement attribute sets the SQL_DESC_BIND_TYPE field in the APD header.

---

**SQL_ATTR_PARAM_OPERATION_PTR (ODBC 3.0)**

An SQLUSMALLINT * value that points to an array of SQLUSMALLINT values used to ignore a parameter during execution of an SQL statement. Each value is set to either SQL_PARAM_PROCEED (for the parameter to be executed) or SQL_PARAM_IGNORE (for the parameter to be ignored).

A set of parameters can be ignored during processing by setting the status value in the array pointed to by SQL_DESC_ARRAY_STATUS_PTR in the APD to SQL_PARAM_IGNORE. A set of parameters is processed if its status value is set to SQL_PARAM_PROCEED or if no elements in the array are set.
This statement attribute can be set to a null pointer, in which case the driver does not return parameter status values. This attribute can be set at any time, but the new value is not used until the next time `SQLExecDirect` or `SQLExecute` is called.

This attribute is ignored when there is no bound parameter.

For more information, see Using Arrays of Parameters.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the APD header.

---

### SQL_ATTR_PARAM_STATUS_PTR (ODBC 3.0)

An SQLSMALLINT * value that points to an array of SQLSMALLINT values containing status information for each row of parameter values after a call to `SQLExecute` or `SQLExecDirect`. This field is required only if PARAMSET_SIZE is greater than 1.

The status values can contain the following values:

- **SQL_PARAM_SUCCESS**: The SQL statement was successfully executed for this set of parameters.
- **SQL_PARAM_SUCCESS_WITH_INFO**: The SQL statement was successfully executed for this set of parameters; however, warning information is available in the diagnostics data structure.
- **SQL_PARAM_ERROR**: There was an error in processing this set of parameters. Additional error information is available in the diagnostics data structure.
- **SQL_PARAM_UNUSED**: This parameter set was unused, possibly due to the fact that some previous parameter set caused an error that aborted further processing, or because SQL_PARAM_IGNORE was set for that set of parameters in the array specified by the SQL_ATTR_PARAM_OPERATION_PTR.
- **SQL_PARAM_DIAG_UNAVAILABLE**: The driver treats arrays of parameters as a monolithic unit and so does not generate this level of error information.

This statement attribute can be set to a null pointer, in which case the driver does not return parameter status values. This attribute can be set at any time, but the new value is not used until the next time `SQLExecute` or `SQLExecDirect` is called. Note that setting this attribute can affect the output parameter behavior implemented by the driver.

For more information, see Using Arrays of Parameters.

Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the IPD header.
| SQL_ATTR_PARAMS_PROCESSED_PTR (ODBC 3.0) | An SQLULEN * record field that points to a buffer in which to return the number of sets of parameters that have been processed, including error sets. No number will be returned if this is a null pointer. 

Setting this statement attribute sets the SQL_DESC_ROWS_PROCESSED_PTR field in the IPD header. 

If the call to **SQLExecDirect** or **SQLExecute** that fills in the buffer pointed to by this attribute does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined. 

For more information, see Using Arrays of Parameters. |
| SQL_ATTR_PARAMSET_SIZE (ODBC 3.0) | An SQLULEN value that specifies the number of values for each parameter. If SQL_ATTR_PARAMSET_SIZE is greater than 1, SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR of the APD point to arrays. The cardinality of each array is equal to the value of this field. 

This attribute is ignored when there is no bound parameter. 

For more information, see Using Arrays of Parameters. 

Setting this statement attribute sets the SQL_DESC_ARRAY_SIZE field in the APD header. |
| SQL_ATTR_QUERY_TIMEOUT (ODBC 1.0) | An SQLULEN value corresponding to the number of seconds to wait for an SQL statement to execute before returning to the application. If ValuePtr is equal to 0 (default), there is no timeout. 

If the specified timeout exceeds the maximum timeout in the data source or is smaller than the minimum timeout, **SQLSetStmtAttr** substitutes that value and returns SQLSTATE 01S02 (Option value changed). 

Note that the application need not call **SQLCloseCursor** to reuse the statement if a SELECT statement timed out. 

The query timeout set in this statement attribute is valid in both synchronous and asynchronous modes. |
| SQL_ATTR_RETRIEVE_DATA (ODBC 2.0) | An SQLULEN value: 

SQL_RD_ON = **SQLFetchScroll** and, in ODBC 3.x, **SQLFetch** retrieve data after it positions the cursor to the specified location. This is the default. 

SQL_RD_OFF = **SQLFetchScroll** and, in ODBC 3.x, |
**SQLFetch** do not retrieve data after it positions the cursor.

By setting SQL_RETRIEVE_DATA to SQL_RD_OFF, an application can verify that a row exists or retrieve a bookmark for the row without incurring the overhead of retrieving rows. For more information, see *Scrolling and Fetching Rows*.

The value of this attribute can be set on an open cursor; however, the setting might not take effect immediately, in which case the driver will return SQLSTATE 01S02 (Option value changed) and reset the attribute to its original value.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_ROW_ARRAY_SIZE (ODBC 3.0)</td>
<td>An SQLULEN value that specifies the number of rows returned by each call to <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong>. It is also the number of rows in a bookmark array used in a bulk bookmark operation in <strong>SQLBulkOperations</strong>. The default value is 1. If the specified rowset size exceeds the maximum rowset size supported by the data source, the driver substitutes that value and returns SQLSTATE 01S02 (Option value changed). For more information, see <em>Rowset Size</em>. Setting this statement attribute sets the SQL_DESC_ARRAY_SIZE field in the ARD header.</td>
</tr>
<tr>
<td>SQL_ATTR_ROW_BIND_OFFSET_PTR (ODBC 3.0)</td>
<td>An SQLULEN * value that points to an offset added to pointers to change binding of column data. If this field is non-null, the driver dereferences the pointer, adds the dereferenced value to each of the deferred fields in the descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR), and uses the new pointer values when binding. It is set to null by default. Setting this statement attribute sets the SQL_DESC_BIND_OFFSET_PTR field in the ARD header.</td>
</tr>
<tr>
<td>SQL_ATTR_ROW_BIND_TYPE (ODBC 1.0)</td>
<td>An SQLULEN value that sets the binding orientation to be used when <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong> is called on the associated statement. Column-wise binding is selected by setting the value to SQL_BIND_BY_COLUMN. Row-wise binding is selected by setting the value to the length of a structure or an instance of a buffer into which result columns will be bound. If a length is specified, it must include space for all of the bound columns and any padding of the structure or buffer to ensure that when the address of a bound column is incremented with the specified length, the result will point to the beginning of the same column in the next row. When using the <strong>sizeof</strong> operator with structures or unions</td>
</tr>
</tbody>
</table>
in ANSI C, this behavior is guaranteed.

Column-wise binding is the default binding orientation for **SQLFetch** and **SQLFetchScroll**.

For more information, see [Binding Columns for Use with Block Cursors](#).

Setting this statement attribute sets the SQL_DESC_BIND_TYPE field in the ARD header.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_ROW_NUMBER (ODBC 2.0)</td>
<td>An SQLULEN value that is the number of the current row in the entire result set. If the number of the current row cannot be determined or there is no current row, the driver returns 0. This attribute can be retrieved by a call to SQLGetStmtAttr but not set by a call to SQLSetStmtAttr.</td>
</tr>
<tr>
<td>SQL_ATTR_ROW_OPERATION_PTR (ODBC 3.0)</td>
<td>An SQLUSMALLINT * value that points to an array of SQLUSMALLINT values used to ignore a row during a bulk operation using SQLSetPos. Each value is set to either SQL_ROW_PROCEED (for the row to be included in the bulk operation) or SQL_ROW_IGNORE (for the row to be excluded from the bulk operation). (Rows cannot be ignored by using this array during calls to SQLBulkOperations.) This statement attribute can be set to a null pointer, in which case the driver does not return row status values. This attribute can be set at any time, but the new value is not used until the next time SQLSetPos is called. For more information, see <a href="#">Updating Rows in the Rowset with SQLSetPos</a> and <a href="#">Deleting Rows in the Rowset with SQLSetPos</a>. Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the ARD.</td>
</tr>
<tr>
<td>SQL_ATTR_ROW_STATUS_PTR (ODBC 3.0)</td>
<td>An SQLUSMALLINT * value that points to an array of SQLUSMALLINT values containing row status values after a call to SQLFetch or SQLFetchScroll. The array has as many elements as there are rows in the rowset. This statement attribute can be set to a null pointer, in which case the driver does not return row status values. This attribute can be set at any time, but the new value is not used until the next time SQLBulkOperations, SQLFetch, SQLFetchScroll, or SQLSetPos is called. For more information, see <a href="#">Number of Rows Fetched and Status</a>. Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the ARD.</td>
</tr>
</tbody>
</table>
**SQL_DESC_ARRAY_STATUS_PTR field in the IRD header.**

This attribute is mapped by an ODBC 2.x driver to the rgbRowStatus array in a call to **SQLExtendedFetch.**

| SQL_ATTR_ROWS_FETCHED_PTR (ODBC 3.0) | An SQLULEN * value that points to a buffer in which to return the number of rows fetched after a call to **SQLFetch** or **SQLFetchScroll**; the number of rows affected by a bulk operation performed by a call to **SQLSetPos** with an Operation argument of SQL_REFRESH; or the number of rows affected by a bulk operation performed by **SQLBulkOperations**. This number includes error rows.

For more information, see Number of Rows Fetched and Status.

Setting this statement attribute sets the SQL_DESC_ROWS_PROCESSED_PTR field in the IRD header.

If the call to **SQLFetch** or **SQLFetchScroll** that fills in the buffer pointed to by this attribute does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

| SQL_ATTR_SIMULATE_CURSOR (ODBC 2.0) | An SQLULEN value that specifies whether drivers that simulate positioned update and delete statements guarantee that such statements affect only one single row.

To simulate positioned update and delete statements, most drivers construct a searched UPDATE or DELETE statement containing a WHERE clause that specifies the value of each column in the current row. Unless these columns make up a unique key, such a statement can affect more than one row.

To guarantee that such statements affect only one row, the driver determines the columns in a unique key and adds these columns to the result set. If an application guarantees that the columns in the result set make up a unique key, the driver is not required to do so. This may reduce execution time.

**SQL_SC_NON_UNIQUE** = The driver does not guarantee that simulated positioned update or delete statements will affect only one row; it is the application’s responsibility to do so. If a statement affects more than one row, **SQLExecute**, **SQLExecDirect**, or **SQLSetPos** returns SQLSTATE 01001 (Cursor operation conflict).

**SQL_SC_TRY_UNIQUE** = The driver attempts to guarantee that simulated positioned update or delete statements affect only one row. The driver always executes such statements, even if they might affect more than one row,
such as when there is no unique key. If a statement affects more than one row, *SQLExecute, SQLExecDirect*, or *SQLSetPos* returns SQLSTATE 01001 (Cursor operation conflict).

SQL_SC_UNIQUE = The driver guarantees that simulated positioned update or delete statements affect only one row. If the driver cannot guarantee this for a given statement, *SQLExecDirect* or *SQLPrepare* returns an error.

If the data source provides native SQL support for positioned update and delete statements and the driver does not simulate cursors, SQL_SUCCESS is returned when SQL_SC_UNIQUE is requested for SQL_SIMULATE_CURSOR. SQL_SUCCESS_WITH_INFO is returned if SQL_SC_TRY_UNIQUE or SQL_SC_NON_UNIQUE is requested. If the data source provides the SQL_SC_TRY_UNIQUE level of support and the driver does not, SQL_SUCCESS is returned for SQL_SC_TRY_UNIQUE and SQL_SUCCESS_WITH_INFO is returned for SQL_SC_NON_UNIQUE.

If the specified cursor simulation type is not supported by the data source, the driver substitutes a different simulation type and returns SQLSTATE 01S02 (Option value changed). For SQL_SC_UNIQUE, the driver substitutes, in order, SQL_SC_TRY_UNIQUE or SQL_SC_NON_UNIQUE. For SQL_SC_TRY_UNIQUE, the driver substitutes SQL_SC_NON_UNIQUE.

The default is SQL_SC_UNIQUE.

For more information, see [Simulating Positioned Update and Delete Statements](#).

<table>
<thead>
<tr>
<th>SQL_ATTR_USE_BOOKMARKS (ODBC 2.0)</th>
<th>An SQUALEN value that specifies whether an application will use bookmarks with a cursor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_UB_OFF = Off (the default)</td>
<td></td>
</tr>
<tr>
<td>SQL_UB_VARIABLE = An application will use bookmarks with a cursor, and the driver will provide variable-length bookmarks if they are supported. SQL_UB_FIXED is deprecated in ODBC 3.x. ODBC 3.x applications should always use variable-length bookmarks, even when working with ODBC 2.x drivers (which supported only 4-byte, fixed-length bookmarks). This is because a fixed-length bookmark is just a special case of a variable-length bookmark. When working with an ODBC 2.x driver, the Driver Manager maps SQL_UB_VARIABLE to SQL_UB_FIXED.</td>
<td></td>
</tr>
</tbody>
</table>

To use bookmarks with a cursor, the application must specify this attribute with the SQL_UB_VARIABLE value before opening the cursor.
See Column-Wise Binding and Row-Wise Binding.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Returning the setting of a connection attribute</td>
<td>SQLGetConnectAttr Function</td>
</tr>
<tr>
<td>Returning the setting of a statement attribute</td>
<td>SQLGetStmtAttr Function</td>
</tr>
<tr>
<td>Setting a connection attribute</td>
<td>SQLSetConnectAttr Function</td>
</tr>
<tr>
<td>Setting a single field of the descriptor</td>
<td>SQLSetDescField Function</td>
</tr>
</tbody>
</table>

**See Also**

ODBC API Reference  
ODBC Header Files

© 2016 Microsoft

**SQLSetStmtOption Function**

**Conformance**  
Version Introduced: ODBC 1.0  
Standards Compliance: Deprecated

**Summary**  
In ODBC 3.x, the ODBC 2.0 function `SQLSetStmtOption` has been replaced by `SQLSetStmtAttr`. For more information, see `SQLSetStmtAttr`.

**Note**

For more information about what the Driver Manager maps this function to when an ODBC 2.x application is working with an ODBC 3.x driver, see Mapping Deprecated Functions in Appendix G: Driver Guidelines for Backward Compatibility.
Remarks

See ODBC 64-Bit Information, if your application will run on a 64-bit operating system.

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLSpecialColumns Function

Conformance
Version Introduced: ODBC 1.0
Standards Compliance: Open Group

Summary

SQLSpecialColumns retrieves the following information about columns within a specified table:

- The optimal set of columns that uniquely identifies a row in the table.
- Columns that are automatically updated when any value in the row is updated by a transaction.

Syntax

```c
SQLRETURN SQLSpecialColumns(
    SQLHSTMT StatementHandle,
    SQLSMALLINT IdentifierType,
    SQLCHAR * CatalogName,
    SQLSMALLINT NameLength1,
    SQLCHAR * SchemaName,
    SQLSMALLINT NameLength2,
    SQLCHAR * TableName,
    SQLSMALLINT NameLength3,
    SQLSMALLINT Scope,
    SQLSMALLINT Nullable);
```

Arguments

StatementHandle
[Input] Statement handle.
**IdentifierType**

[Input] Type of column to return. Must be one of the following values:

SQL_BEST_ROWID: Returns the optimal column or set of columns that, by retrieving values from the column or columns, allows any row in the specified table to be uniquely identified. A column can be either a pseudo-column specifically designed for this purpose (as in Oracle ROWID or Ingres TID) or the column or columns of any unique index for the table.

SQL_ROWVER: Returns the column or columns in the specified table, if any, that are automatically updated by the data source when any value in the row is updated by any transaction (as in SQLBase ROWID or Sybase TIMESTAMP).

**CatalogName**

[Input] Catalog name for the table. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string (""") denotes those tables that do not have catalogs. CatalogName cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, CatalogName is treated as an identifier and its case is not significant. If it is SQL_FALSE, CatalogName is an ordinary argument; it is treated literally, and its case is significant. For more information, see Arguments in Catalog Functions.

**NameLength1**

[Input] Length in characters of *CatalogName.

**SchemaName**

[Input] Schema name for the table. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string (""") denotes those tables that do not have schemas. SchemaName cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, SchemaName is treated as an identifier and its case is not significant. If it is SQL_FALSE, SchemaName is an ordinary argument; it is treated literally, and its case is significant.

**NameLength2**

[Input] Length in characters of *SchemaName.

**TableName**

[Input] Table name. This argument cannot be a null pointer. TableName cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, TableName is treated as an identifier and its case is not significant. If it is SQL_FALSE, TableName is an ordinary argument; it is treated literally, and its case is significant.

**NameLength3**

[Input] Length in characters of *TableName.

**Scope**

[Input] Minimum required scope of the rowid. The returned rowid may be of greater scope. Must be one of the following:

SQL_SCOPE_CURROW: The rowid is guaranteed to be valid only while positioned on that row. A later reselect using rowid may not return a row if the row was updated or deleted by another transaction.
SQL_SCOPE_TRANSACTION: The rowid is guaranteed to be valid for the duration of the current transaction.

SQL_SCOPE_SESSION: The rowid is guaranteed to be valid for the duration of the session (across transaction boundaries).

Nullable

[Input] Determines whether to return special columns that can have a NULL value. Must be one of the following:

SQL_NO_NULLS: Exclude special columns that can have NULL values. Some drivers cannot support SQL_NO_NULLS, and these drivers will return an empty result set if SQL_NO_NULLS was specified. Applications should be prepared for this case and request SQL_NO_NULLS only if it is absolutely required.

SQL_NULLABLE: Return special columns even if they can have NULL values.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLSpecialColumns returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLSpecialColumns and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>A cursor was open on the StatementHandle, and SQLFetch or SQLFetchScroll had been called. This error is returned by the Driver Manager if SQLFetch or SQLFetchScroll has not returned SQL_NO_DATA and is returned by the driver if SQLFetch or SQLFetchScroll has returned SQL_NO_DATA. A cursor was open on the StatementHandle, but SQLFetch or SQLFetchScroll had not been called.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure: The transaction was rolled back due to a resource deadlock with another transaction.</td>
<td></td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown: The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>General error: An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <em>MessageText</em> buffer describes the error and its cause.</td>
<td></td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error: The driver was unable to allocate memory required to support execution or completion of the function.</td>
<td></td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled: Asynchronous processing was enabled for the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code>. Then the function was called again on the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code> from a different thread in a multithread application.</td>
<td></td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer: The <code>TableName</code> argument was a null pointer. The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the <code>CatalogName</code> argument was a null pointer, and the SQL_CATALOG_NAME InfoType returns that catalog names are supported. (DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the <code>SchemaName</code> argument was a null pointer.</td>
<td></td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error: (DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This function was still executing when <code>SQLSpecialColumns</code> was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for the <code>StatementHandle</code> and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. (DM) An asynchronously executing function (not this one) was called for the <code>StatementHandle</code> and was still executing when this function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for the <code>StatementHandle</code> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value of one of the length arguments was less than 0 but not equal to SQL_NTS. The value of one of the length arguments exceeded the maximum length value for the corresponding name. The maximum length of each name can be obtained by calling SQLGetInfo with the InfoType values: SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, or SQL_MAX_TABLE_NAME_LEN.</td>
</tr>
<tr>
<td>HY097</td>
<td>Column type out of range</td>
<td>(DM) An invalid IdentifierType value was specified.</td>
</tr>
<tr>
<td>HY098</td>
<td>Scope type out of range</td>
<td>(DM) An invalid Scope value was specified.</td>
</tr>
<tr>
<td>HY099</td>
<td>Nullable type out of range</td>
<td>(DM) An invalid Nullable value was specified.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>A catalog was specified, and the driver or data source does not support catalogs. A schema was specified, and the driver or data source does not support schemas. The combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes was not supported by the driver or data source. The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the requested result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is</td>
</tr>
</tbody>
</table>
set through **SQLSetConnectAttr**, **SQL_ATTR_CONNECTION_TIMEOUT**.

<table>
<thead>
<tr>
<th>IM001</th>
<th>Driver does not support this function</th>
<th>(DM) The driver associated with the <em>StatementHandle</em> does not support the function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><strong>SQLCompleteAsync</strong> has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns <strong>SQL_STILL_EXECUTING</strong> and if notification mode is enabled, <strong>SQLCompleteAsync</strong> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

### Comments

When the *IdentifierType* argument is **SQL_BEST_ROWID**, **SQLSpecialColumns** returns the column or columns that uniquely identify each row in the table. These columns can always be used in a **select-list** or **WHERE** clause. **SQLColumns**, which is used to return a variety of information on the columns of a table, does not necessarily return the columns that uniquely identify each row, or columns that are automatically updated when any value in the row is updated by a transaction. For example, **SQLColumns** might not return the Oracle pseudo-column **ROWID**. This is why **SQLSpecialColumns** is used to return these columns. For more information, see **Uses of Catalog Data**.

**Note**

For more information about the general use, arguments, and returned data of ODBC catalog functions, see **Catalog Functions**.

If there are no columns that uniquely identify each row in the table, **SQLSpecialColumns** returns a rowset with no rows; a subsequent call to **SQLFetch** or **SQLFetchScroll** on the statement returns **SQL_NO_DATA**.

If the *IdentifierType*, *Scope*, or *Nullable* arguments specify characteristics that are not supported by the data source, **SQLSpecialColumns** returns an empty result set.

If the **SQL_ATTR_METADATA_ID** statement attribute is set to **SQL_TRUE**, the **CatalogName**, **SchemaName**, and **TableName** arguments are treated as identifiers, so they cannot be set to a null pointer in certain situations. (For more information, see **Arguments in Catalog Functions**.)

**SQLSpecialColumns** returns the results as a standard result set, ordered by **SCOPE**.

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.
To determine the actual length of the COLUMN_NAME column, an application can call `SQLGetInfo` with the SQL_MAX_COLUMN_NAME_LEN option.

The following table lists the columns in the result set. Additional columns beyond column 8 (PSEUDO_COLUMN) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set rather than specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOPE (ODBC 1.0)</td>
<td>1</td>
<td>Smallint</td>
<td>Actual scope of the rowid. Contains one of the following values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SQL_SCOPE_CURROW SQL_SCOPE_TRANSACTION SQL_SCOPE_SESSION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NULL is returned when IdentifierType is SQL_ROWVER. For a description of each value, see the description of Scope in &quot;Syntax,&quot; earlier in this section.</td>
</tr>
<tr>
<td>COLUMN_NAME (ODBC 1.0)</td>
<td>2</td>
<td>Varchar not NULL</td>
<td>Column name. The driver returns an empty string for a column that does not have a name.</td>
</tr>
<tr>
<td>DATA_TYPE (ODBC 1.0)</td>
<td>3</td>
<td>Smallint not NULL</td>
<td>SQL data type. This can be an ODBC SQL data type or a driver-specific SQL data type. For a list of valid ODBC SQL data types, see SQL Data Types. For information about driver-specific SQL data types, see the driver's documentation.</td>
</tr>
<tr>
<td>TYPE_NAME (ODBC 1.0)</td>
<td>4</td>
<td>Varchar not NULL</td>
<td>Data source–dependent data type name; for example, &quot;CHAR&quot;, &quot;VARCHAR&quot;, &quot;MONEY&quot;, &quot;LONG VARBINARY&quot;, or &quot;CHAR ( ) FOR BIT DATA&quot;.</td>
</tr>
<tr>
<td>COLUMN_SIZE (ODBC 1.0)</td>
<td>5</td>
<td>Integer</td>
<td>The size of the column on the data source. For more information concerning column size, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size.</td>
</tr>
<tr>
<td>BUFFER_LENGTH (ODBC 1.0)</td>
<td>6</td>
<td>Integer</td>
<td>The length in bytes of data transferred on an <code>SQLGetData</code> or <code>SQLFetch</code> operation if SQL_C_DEFAULT is specified. For numeric data, this size may be different than the size of the data stored on the data source. This value is the same as the COLUMN_SIZE column for character or binary data. For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size.</td>
</tr>
</tbody>
</table>
DECIMAL_DIGITS (ODBC 1.0)  |  7  | Smallint  | The decimal digits of the column on the data source. NULL is returned for data types where decimal digits are not applicable. For more information concerning decimal digits, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size.

PSEUDO_COLUMN (ODBC 2.0)  |  8  | Smallint  | Indicates whether the column is a pseudo-column, such as Oracle ROWID:

SQL_PC_UNKNOWN SQL_PC_NOT_PSEUDO SQL_PC_PSEUDO

Note

For maximum interoperability, pseudo-columns should not be quoted with the identifier quote character returned by SQLGetInfo.

After the application retrieves values for SQL_BEST_ROWID, the application can use these values to reselect that row within the defined scope. The SELECT statement is guaranteed to return either no rows or one row.

If an application reselects a row based on the rowid column or columns and the row is not found, the application can assume that the row was deleted or the rowid columns were modified. The opposite is not true: even if the rowid has not changed, the other columns in the row may have changed.

Columns returned for column type SQL_BEST_ROWID are useful for applications that need to scroll forward and back within a result set to retrieve the most recent data from a set of rows. The column or columns of the rowid are guaranteed not to change while positioned on that row.

The column or columns of the rowid may remain valid even when the cursor is not positioned on the row; the application can determine this by checking the SCOPE column in the result set.

Columns returned for column type SQL_ROWVER are useful for applications that need the ability to check whether any columns in a given row have been updated while the row was reselected using the rowid. For example, after reselecting a row using rowid, the application can compare the previous values in the SQL_ROWVER columns to the ones just fetched. If the value in a SQL_ROWVER column differs from the previous value, the application can alert the user that data on the display has changed.

**Code Example**

For a code example of a similar function, see SQLColumns.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLStatistics Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ISO 92

Summary
SQLStatistics retrieves a list of statistics about a single table and the indexes associated with the table. The driver returns the information as a result set.

Syntax

```c
SQLRETURN SQLStatistics(
    SQLHSTMT    StatementHandle,
    SQLCHAR    * CatalogName,
    SQLSMALLINT NameLength1,
    SQLCHAR    * SchemaName,
    SQLSMALLINT NameLength2,
    SQLCHAR    * TableName,
    SQLSMALLINT NameLength3,
    SQLUSMALLINT Unique,
    SQLUSMALLINT Reserved);
```

Arguments
**StatementHandle**

[Input] Statement handle.

**CatalogName**

[Input] Catalog name. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string (""") indicates those tables that do not have catalogs. *CatalogName* cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *CatalogName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *CatalogName* is an ordinary argument; it is treated literally, and its case is significant. For more information, see Arguments in Catalog Functions.

**NameLength1**

[Input] Length in characters of *CatalogName*.

**SchemaName**

[Input] Schema name. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string (""") indicates those tables that do not have schemas. *SchemaName* cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *SchemaName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *SchemaName* is an ordinary argument; it is treated literally, and its case is significant.

**NameLength2**

[Input] Length in characters of *SchemaName*.

**TableName**

[Input] Table name. This argument cannot be a null pointer. *SchemaName* cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *TableName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *TableName* is an ordinary argument; it is treated literally, and its case is significant.

**NameLength3**

[Input] Length in characters of *TableName*.

**Unique**

[Input] Type of index: SQL_INDEX_UNIQUE or SQL_INDEX_ALL.

**Reserved**

[Input] Indicates the importance of the CARDINALITY and PAGES columns in the result set. The following options affect the return of the CARDINALITY and PAGES columns only; index information is returned even if CARDINALITY and PAGES are not returned.

SQL_ENSURE requests that the driver unconditionally retrieve the statistics. (Drivers that conform only to the Open Group standard and do not support ODBC extensions will not be able to support SQL_ENSURE.)

SQL_QUICK requests that the driver retrieve the CARDINALITY and PAGES only if they are readily available from the server. In this case, the driver does not ensure that the values are current. (Applications that are written to the Open Group standard will always get SQL_QUICK behavior from ODBC 3.x-compliant drivers.)
Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLStatistics returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values typically returned by SQLStatistics and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>A cursor was open on the StatementHandle, and SQLFetch or SQLFetchScroll had been called. This error is returned by the Driver Manager if SQLFetch or SQLFetchScroll has not returned SQL_NO_DATA and is returned by the driver if SQLFetch or SQLFetchScroll has returned SQL_NO_DATA. A cursor was open on the StatementHandle, but SQLFetch or SQLFetchScroll had not been called.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back because of a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
</tbody>
</table>
| HY008    | Operation canceled                   | Asynchronous processing was enabled for the StatementHandle. The function was called, and before it
completed execution, **SQLCancel** or **SQLCancelHandle** was called on the **StatementHandle**, and then the function was called again on the **StatementHandle**.

The function was called, and before it completed execution, **SQLCancel** or **SQLCancelHandle** was called on the **StatementHandle** from a different thread in a multithread application.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>The <strong>TableName</strong> argument was a null pointer. The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the <strong>CatalogName</strong> argument was a null pointer, and the SQL_CATALOG_NAME InfoType returns that catalog names are supported. (DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the <strong>SchemaName</strong> argument was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the <strong>StatementHandle</strong>. This asynchronous function was still executing when the <strong>SQLStatistics</strong> function was called. (DM) <strong>SQLExecute</strong>, <strong>SQLExecDirect</strong>, or <strong>SQLMoreResults</strong> was called for the <strong>StatementHandle</strong> and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. (DM) An asynchronously executing function (not this one) was called for the <strong>StatementHandle</strong> and was still executing when this function was called. (DM) <strong>SQLExecute</strong>, <strong>SQLExecDirect</strong>, <strong>SQLBulkOperations</strong>, or <strong>SQLSetPos</strong> was called for the <strong>StatementHandle</strong> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value of one of the name length arguments was less than 0 but not equal to SQL_NTS. The value of one of the name length arguments exceeded the maximum length value for the corresponding name.</td>
</tr>
<tr>
<td>HY100</td>
<td>Uniqueness option type out of range</td>
<td>(DM) An invalid <strong>Unique</strong> value was specified.</td>
</tr>
<tr>
<td>HY101</td>
<td>Accuracy option type out of range</td>
<td>(DM) An invalid <strong>Reserved</strong> value was specified.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>A catalog was specified, and the driver or data source does not support catalogs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A schema was specified, and the driver or data source does not support schemas.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The combination of the current settings of the SQL_ATTR_CONCURREN CY and SQLATTR_CURSOR_TYPE statement attributes was not supported by the driver or data source.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the requested result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td>SQLCompleteAsync has not been called to complete the previous asynchronous operation on this handle.</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, SQLCompleteAsync must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>

**Comments**

**SQLStatistics** returns information about a single table as a standard result set, ordered by NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME, and ORDINAL_POSITION. The result set
combines statistics information (in the CARDINALITY and PAGES columns of the result set) for the table with information about each index. For information about how this information might be used, see Uses of Catalog Data.

To determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and COLUMN_NAME columns, an application can call SQLGetInfo with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

Note
For more information about the general use, arguments, and returned data of ODBC catalog functions, see Catalog Functions.

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>TABLE_CAT</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>TABLE_SCHEM</td>
</tr>
<tr>
<td>SEQ_IN_INDEX</td>
<td>ORDINAL_POSITION</td>
</tr>
<tr>
<td>COLLATION</td>
<td>ASC_OR_DESC</td>
</tr>
</tbody>
</table>

The following table lists the columns in the result set. Additional columns beyond column 13 (FILTER_CONDITION) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set instead of specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_CAT (ODBC 1.0)</td>
<td>1</td>
<td>Varchar</td>
<td>Catalog name of the table to which the statistic or index applies; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those tables that do not have catalogs.</td>
</tr>
<tr>
<td>TABLE_SCHEM (ODBC 1.0)</td>
<td>2</td>
<td>Varchar</td>
<td>Schema name of the table to which the statistic or index applies; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those tables that do not have schemas.</td>
</tr>
<tr>
<td>TABLE_NAME (ODBC 1.0)</td>
<td>3</td>
<td>Varchar</td>
<td>Table name of the table to which the statistic or index applies.</td>
</tr>
<tr>
<td>COLUMN NAME</td>
<td>DATA TYPE</td>
<td>DESCRIPTION</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>NON_UNIQUE (ODBC 1.0)</td>
<td>Smallint</td>
<td>Indicates whether the index does not allow duplicate values: SQL_TRUE if the index values can be nonunique. SQL_FALSE if the index values must be unique. NULL is returned if TYPE is SQL_TABLE_STAT.</td>
<td></td>
</tr>
<tr>
<td>INDEX_QUALIFIER (ODBC 1.0)</td>
<td>Varchar</td>
<td>The identifier that is used to qualify the index name doing a DROP INDEX; NULL is returned if an index qualifier is not supported by the data source or if TYPE is SQL_TABLE_STAT. If a non-null value is returned in this column, it must be used to qualify the index name on a DROP INDEX statement; otherwise, the TABLE_SCHEM should be used to qualify the index name.</td>
<td></td>
</tr>
<tr>
<td>INDEX_NAME (ODBC 1.0)</td>
<td>Varchar</td>
<td>Index name; NULL is returned if TYPE is SQL_TABLE_STAT.</td>
<td></td>
</tr>
<tr>
<td>TYPE (ODBC 1.0)</td>
<td>Smallint not NULL</td>
<td>Type of information being returned: SQL_TABLE_STAT indicates a statistic for the table (in the CARDINALITY or PAGES column). SQL_INDEX_BTREE indicates a B-Tree index. SQL_INDEX_CLUSTERS indicates a clustered index. SQL_INDEX_CONTENT indicates a content index. SQL_INDEX_HASHED indicates a hashed index. SQL_INDEX_OTHER indicates another type of index.</td>
<td></td>
</tr>
<tr>
<td>ORDINAL_POSITION (ODBC 1.0)</td>
<td>Smallint</td>
<td>Column sequence number in index (starting with 1); NULL is returned if TYPE is SQL_TABLE_STAT.</td>
<td></td>
</tr>
<tr>
<td>COLUMN_NAME (ODBC 1.0)</td>
<td>Varchar</td>
<td>Column name. If the column is based on an expression, such as SALARY + BENEFITS, the expression is returned; if the expression cannot be determined, an empty string is returned. NULL is returned if TYPE is SQL_TABLE_STAT.</td>
<td></td>
</tr>
<tr>
<td>ASC_OR_DESC (ODBC 1.0)</td>
<td>Char(1)</td>
<td>Sort sequence for the column: &quot;A&quot; for ascending; &quot;D&quot; for descending; NULL is returned if column sort sequence is not supported by the data source or if TYPE is SQL_TABLE_STAT.</td>
<td></td>
</tr>
<tr>
<td>CARDINALITY (ODBC 1.0)</td>
<td>Integer</td>
<td>Cardinality of table or index; number of rows in table if TYPE is SQL_TABLE_STAT; number of unique values in the index if TYPE is not SQL_TABLE_STAT; NULL is returned if the value is not available from the data</td>
<td></td>
</tr>
</tbody>
</table>
### PAGES (ODBC 1.0)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td>Number of pages used to store the index or table; number of pages for the table if TYPE is SQL_TABLE_STAT; number of pages for the index if TYPE is not SQL_TABLE_STAT; NULL is returned if the value is not available from the data source or if not applicable to the data source.</td>
</tr>
</tbody>
</table>

### FILTER_CONDITION (ODBC 2.0)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td></td>
<td>If the index is a filtered index, this is the filter condition, such as SALARY &gt; 30000; if the filter condition cannot be determined, this is an empty string. NULL if the index is not a filtered index, it cannot be determined whether the index is a filtered index, or TYPE is SQL_TABLE_STAT.</td>
</tr>
</tbody>
</table>

If the row in the result set corresponds to a table, the driver sets TYPE to SQL_TABLE_STAT and sets NON_UNIQUE, INDEX_QUALIFIER, INDEX_NAME, ORDINAL_POSITION, COLUMN_NAME, and ASC_OR_DESC to NULL. If CARDINALITY or PAGES are not available from the data source, the driver sets them to NULL.

## Code Example

For a code example of a similar function, see [SQLColumns](#).

## Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction.</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Returning the columns of foreign keys</td>
<td>SQLForeignKeys Function</td>
</tr>
<tr>
<td>Returning the columns of a primary key</td>
<td>SQLPrimaryKeys Function</td>
</tr>
</tbody>
</table>

## See Also

[ODBC API Reference](#)
SQLTablePrivileges Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: ODBC

Summary
SQLTablePrivileges returns a list of tables and the privileges associated with each table. The driver returns the information as a result set on the specified statement.

Syntax

```
SQLRETURN SQLTablePrivileges(
    SQLHSTMT StatementHandle,
    SQLCHAR * CatalogName,
    SQLSMALLINT NameLength1,
    SQLCHAR * SchemaName,
    SQLSMALLINT NameLength2,
    SQLCHAR * TableName,
    SQLSMALLINT NameLength3);
```

Arguments

**StatementHandle**
[Input] Statement handle.

**CatalogName**
[Input] Table catalog. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have catalogs. CatalogName cannot contain a string search pattern.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, CatalogName is treated as an identifier and its case is not significant. If it is SQL_FALSE, CatalogName is an ordinary argument; it is treated literally, and its case is significant. For more information, see Arguments in Catalog Functions.

**NameLength1**
[Input] Length in characters of *CatalogName.

**SchemaName**
[Input] String search pattern for schema names. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") denotes those tables that do not have schemas.
If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, SchemaName is treated as an identifier and its case is not significant. If it is SQL_FALSE, SchemaName is a pattern value argument; it is treated literally, and its case is significant.

NameLength2
[Input] Length in characters of *SchemaName.

TableName
[Input] String search pattern for table names.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, TableName is treated as an identifier and its case is not significant. If it is SQL_FALSE, TableName is a pattern value argument; it is treated literally, and its case is significant.

NameLength3
[Input] Length in characters of *TableName.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

When SQLTablePrivileges returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLTablePrivileges and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>A cursor was open on the StatementHandle, and SQLFetch or SQLFetchScroll had been called. This error is returned by the Driver Manager if SQLFetch or SQLFetchScroll has not returned SQL_NO_DATA and is returned by the driver if SQLFetch or SQLFetchScroll has returned SQL_NO_DATA. A cursor was open on the StatementHandle, but SQLFetch or SQLFetchScroll had not been called.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Message</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back due to a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the StatementHandle. The SQLTablePrivileges function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle. Then the SQLTablePrivileges function was called again on the StatementHandle. The SQLTablePrivileges function was called, and before it completed execution, SQLCancel or SQLCancelHandle was called on the StatementHandle from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the CatalogName argument was a null pointer, and the SQL_CATALOG_NAME InfoType returns that catalog names are supported. (DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the SchemaName or TableName argument was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the StatementHandle. This asynchronous function was still executing when the SQLTablePrivileges function was called. (DM) SQLExecute, SQLExecDirect, or SQLMoreResults was called for the StatementHandle and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. (DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called. (DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
</tbody>
</table>
| HY090  | Invalid string or buffer length                       | (DM) The value of one of the name length arguments was less than 0 but not equal to SQL_NTS.  
The value of one of the name length arguments exceeded the maximum length value for the corresponding qualifier or name. |
| HY117  | Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed. | (DM) For more information about the suspended state, see SQLEndTran Function. |
| HYC00  | Optional feature not implemented                      | A catalog was specified, and the driver or data source does not support catalogs.  
A schema was specified, and the driver or data source does not support schemas.  
A string search pattern was specified for the table schema, table name, or column name, and the data source does not support search patterns for one or more of those arguments.  
The combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes was not supported by the driver or data source.  
The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks. |
| HYT00  | Timeout expired                                       | The query timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT. |
| HYT01  | Connection timeout expired                            | The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT. |
| IM001  | Driver does not support this function                 | (DM) The driver associated with the StatementHandle does not support the function. |
| IM017  | Polling is disabled in asynchronous notification mode | Whenever the notification model is used, polling is disabled. |
### Comments

The `SchemaName` and `TableName` arguments accept search patterns. For more information about valid search patterns, see [Pattern Value Arguments](#).

`SQLTablePrivileges` returns the results as a standard result set, ordered by `TABLE_CAT`, `TABLE_SCHEM`, `TABLE_NAME`, `PRIVILEGE`, and `GRANTEE`.

To determine the actual lengths of the `TABLE_CAT`, `TABLE_SCHEM`, and `TABLE_NAME` columns, an application can call `SQLGetInfo` with the `SQL_MAX_CATALOG_NAME_LEN`, `SQL_MAX_SCHEMA_NAME_LEN`, and `SQL_MAX_TABLE_NAME_LEN` options.

#### Note

For more information about the general use, arguments, and returned data of ODBC catalog functions, see [Catalog Functions](#).

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>TABLE_CAT</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>TABLE_SCHEM</td>
</tr>
</tbody>
</table>

The following table lists the columns in the result set. Additional columns beyond column 7 (IS_GRANTABLE) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set rather than specifying an explicit ordinal position. For more information, see [Data Returned by Catalog Functions](#).

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_CAT (ODBC 1.0)</td>
<td>1</td>
<td>Varchar</td>
<td>Catalog name; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;) for those tables that do not have catalogs.</td>
</tr>
<tr>
<td><strong>TABLE_SCHEM (ODBC 1.0)</strong></td>
<td>2</td>
<td>Varchar</td>
<td>Schema name; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;&quot;) for those tables that do not have schemas.</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---</td>
<td>---------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td><strong>TABLE_NAME (ODBC 1.0)</strong></td>
<td>3</td>
<td>Varchar</td>
<td>Table name.</td>
</tr>
<tr>
<td><strong>GRANTOR (ODBC 1.0)</strong></td>
<td>4</td>
<td>Varchar</td>
<td>Name of the user who granted the privilege; NULL if not applicable to the data source. For all rows in which the value in the GRANTEE column is the owner of the object, the GRANTOR column will be &quot;_SYSTEM&quot;.</td>
</tr>
<tr>
<td><strong>GRANTEE (ODBC 1.0)</strong></td>
<td>5</td>
<td>Varchar</td>
<td>Name of the user to whom the privilege was granted.</td>
</tr>
<tr>
<td><strong>PRIVILEGE (ODBC 1.0)</strong></td>
<td>6</td>
<td>Varchar</td>
<td>The table privilege. May be one of the following or a data source–specific privilege. SELECT: The grantee is permitted to retrieve data for one or more columns of the table. INSERT: The grantee is permitted to insert new rows containing data for one or more columns into the table. UPDATE: The grantee is permitted to update the data in one or more columns of the table. DELETE: The grantee is permitted to delete rows of data from the table. REFERENCES: The grantee is permitted to refer to one or more columns of the table within a constraint (for example, a unique, referential, or table check constraint). The scope of action permitted the grantee by a given table privilege is data source–dependent. For example, the UPDATE privilege might permit the grantee to update all columns in a table on one data source and only those columns for which the grantor has the UPDATE privilege on another data source.</td>
</tr>
<tr>
<td><strong>IS_GRANTABLE (ODBC 1.0)</strong></td>
<td>7</td>
<td>Varchar</td>
<td>Indicates whether the grantee is permitted to grant the privilege to other users; &quot;YES&quot;, &quot;NO&quot;, or NULL if unknown or not applicable to the data source. A privilege is either grantable or not grantable but not both. The result set returned by <strong>SQLColumnPrivileges</strong> will never contain two rows for which all columns except the <strong>IS_GRANTABLE</strong> column contain the same value.</td>
</tr>
</tbody>
</table>
Code Example
For a code example of a similar function, see SQLColumns.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Returning privileges for a column or columns</td>
<td>SQLColumnPrivileges Function</td>
</tr>
<tr>
<td>Returning the columns in a table or tables</td>
<td>SQLColumns Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Returning table statistics and indexes</td>
<td>SQLStatistics Function</td>
</tr>
<tr>
<td>Returning a list of tables in a data source</td>
<td>SQLTables Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLTables Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Open Group

Summary
SQLTables returns the list of table, catalog, or schema names, and table types, stored in a specific data source. The driver returns the information as a result set.
Syntax

```c
SQLRETURN SQLTables(
    SQLHSTMT  StatementHandle,
    SQLCHAR *  CatalogName,
    SQLSMALLINT NameLength1,
    SQLCHAR *  SchemaName,
    SQLSMALLINT NameLength2,
    SQLCHAR *  TableName,
    SQLSMALLINT NameLength3,
    SQLCHAR *  TableType,
    SQLSMALLINT NameLength4);
```

Arguments

**StatementHandle**

[Input] Statement handle for retrieved results.

**CatalogName**

[Input] Catalog name. The *CatalogName* argument accepts search patterns if the SQL_ODBC_VERSION environment attribute is SQL_OV_ODBC3; it does not accept search patterns if SQL_OV_ODBC2 is set. If a driver supports catalogs for some tables but not for others, such as when a driver retrieves data from different DBMSs, an empty string ("") indicates those tables that do not have catalogs.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *CatalogName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *CatalogName* is a pattern value argument; it is treated literally, and its case is significant. For more information, see Arguments in Catalog Functions.

**NameLength1**

[Input] Length in characters of *CatalogName*.

**SchemaName**

[Input] String search pattern for schema names. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, an empty string ("") indicates those tables that do not have schemas.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *SchemaName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *SchemaName* is a pattern value argument; it is treated literally, and its case is significant.

**NameLength2**

[Input] Length in characters of *SchemaName*.

**TableName**

[Input] String search pattern for table names.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, *TableName* is treated as an identifier and its case is not significant. If it is SQL_FALSE, *TableName* is a pattern value argument; it is treated literally, and its case is significant.
**TableName**

[Input] Length in characters of *TableName.

**TableType**

[Input] List of table types to match.

Notice that the SQL_ATTR_METADATA_ID statement attribute has no effect upon the **TableType** argument. **TableType** is a value list argument, regardless of the setting of SQL_ATTR_METADATA_ID.

**NameLength**

[Input] Length in characters of *TableType.

## Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

## Diagnostics

When **SQLTables** returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling **SQLGetDiagRec** with a **HandleType** of SQL_HANDLE_STMT and a **Handle** of **StatementHandle**. The following table lists the SQLSTATE values typically returned by **SQLTables** and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)</td>
</tr>
<tr>
<td>08S01</td>
<td>Communication link failure</td>
<td>The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>A cursor was open on the <strong>StatementHandle</strong>, and <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong> had been called. This error is returned by the Driver Manager if <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong> has not returned SQL_NO_DATA and is returned by the driver if <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong> has returned SQL_NO_DATA. A cursor was open on the <strong>StatementHandle</strong>, but <strong>SQLFetch</strong> or <strong>SQLFetchScroll</strong> had not been called.</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>The transaction was rolled back because of a resource deadlock with another transaction.</td>
</tr>
<tr>
<td>Code</td>
<td>Message</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by <code>SQLGetDiagRec</code> in the <em>MessageText</em> buffer describes the error and its cause.</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
<td>The driver was unable to allocate memory that is required to support execution or completion of the function.</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
<td>Asynchronous processing was enabled for the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code>. Then the function was called again on the <code>StatementHandle</code>. The function was called, and before it completed execution, <code>SQLCancel</code> or <code>SQLCancelHandle</code> was called on the <code>StatementHandle</code> from a different thread in a multithread application.</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, the CatalogName argument was a null pointer, and the SQL_CATALOG_NAME InfoType returns that catalog names are supported. (DM) The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and the SchemaName or TableName argument was a null pointer.</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>(DM) An asynchronously executing function was called for the connection handle that is associated with the <code>StatementHandle</code>. This asynchronous function was still executing when SQLTables was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, or <code>SQLMoreResults</code> was called for the <code>StatementHandle</code> and returned SQL_PARAM_DATA_AVAILABLE. This function was called before data was retrieved for all streamed parameters. (DM) An asynchronously executing function (not this one) was called for the <code>StatementHandle</code> and was still executing when this function was called. (DM) <code>SQLExecute</code>, <code>SQLExecDirect</code>, <code>SQLBulkOperations</code>, or <code>SQLSetPos</code> was called for the <code>StatementHandle</code> and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>(DM) The value of one of the length arguments was less than 0 but not equal to SQL_NTS. The value of one of the name length arguments exceeded the maximum length value for the corresponding name.</td>
</tr>
<tr>
<td>HY117</td>
<td>Connection is suspended due to unknown transaction state. Only disconnect and read-only functions are allowed.</td>
<td>(DM) For more information about suspended state, see SQLEndTran Function.</td>
</tr>
<tr>
<td>HYC00</td>
<td>Optional feature not implemented</td>
<td>A catalog was specified, and the driver or data source does not support catalogs. A schema was specified, and the driver or data source does not support schemas. A string search pattern was specified for the catalog name, table schema, or table name, and the data source does not support search patterns for one or more of those arguments. The combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes was not supported by the driver or data source. The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the driver does not support bookmarks.</td>
</tr>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>The query timeout period expired before the data source returned the requested result set. The timeout period is set through SQLSetStmtAttr, SQL_ATTR_QUERY_TIMEOUT.</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>(DM) The driver associated with the StatementHandle does not support the function.</td>
</tr>
<tr>
<td>IM017</td>
<td>Polling is disabled in asynchronous notification mode</td>
<td>Whenever the notification model is used, polling is disabled.</td>
</tr>
<tr>
<td>IM018</td>
<td><strong>SQLCompleteAsync</strong> has not been called to complete the previous asynchronous</td>
<td>If the previous function call on the handle returns SQL_STILL_EXECUTING and if notification mode is enabled, <strong>SQLCompleteAsync</strong> must be called on the handle to do post-processing and complete the operation.</td>
</tr>
</tbody>
</table>
Comments

**SQLTables** lists all tables in the requested range. A user may or may not have SELECT privileges to any of these tables. To check accessibility, an application can:

- Call **SQLGetInfo** and check the SQL_ACCESSIBLE_TABLES information type.
- Call **SQLTablePrivileges** to check the privileges for each table.

Otherwise, the application must be able to handle a situation where the user selects a table for which **SELECT** privileges are not granted.

The **SchemaName** and **TableName** arguments accept search patterns. The **CatalogName** argument accepts search patterns if the SQL_ODBC_VERSION environment attribute is SQL_OV_ODBC3; it does not accept search patterns if SQL_OV_ODBC2 is set. If SQL_OV_ODBC3 is set, an ODBC 3.x driver will require that wildcard characters in the **CatalogName** argument be escaped to be treated literally. For more information about valid search patterns, see **Pattern Value Arguments**.

**Note**

For more information about the general use, arguments, and returned data of ODBC catalog functions, see **Catalog Functions**.

To support enumeration of catalogs, schemas, and table types, the following special semantics are defined for the **CatalogName**, **SchemaName**, **TableName**, and **TableType** arguments of **SQLTables**:

- If **CatalogName** is SQL_ALL_CATALOGS and **SchemaName** and **TableName** are empty strings, the result set contains a list of valid catalogs for the data source. (All columns except the TABLE_CAT column contain NULLs.)
- If **SchemaName** is SQL_ALL_SCHEMAS and **CatalogName** and **TableName** are empty strings, the result set contains a list of valid schemas for the data source. (All columns except the TABLE_SCHM column contain NULLs.)
- If **TableType** is SQL_ALL_TABLE_TYPES and **CatalogName**, **SchemaName**, and **TableName** are empty strings, the result set contains a list of valid table types for the data source. (All columns except the TABLE_TYPE column contain NULLs.)

If **TableType** is not an empty string, it must contain a list of comma-separated values for the types of interest; each value can be enclosed in single quotation marks (') or unquoted, for example, 'TABLE', 'VIEW' or TABLE, VIEW. An application should always specify the table type in uppercase; the driver should convert the table type to whatever case is needed by the data source. If the data source does not support a specified table type, **SQLTables** does not return any results for that type.

**SQLTables** returns the results as a standard result set, ordered by TABLE_TYPE, TABLE_CAT, TABLE_SCHM, and TABLE_NAME. For information about how this information might be used, see **Uses of Catalog Data**.
To determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, and TABLE_NAME columns, an application can call `SQLGetInfo` with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, and SQL_MAX_TABLE_NAME_LEN information types.

The following columns have been renamed for ODBC 3.x. The column name changes do not affect backward compatibility because applications bind by column number.

<table>
<thead>
<tr>
<th>ODBC 2.0 column</th>
<th>ODBC 3.x column</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>TABLE_CAT</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>TABLE_SCHEM</td>
</tr>
</tbody>
</table>

The following table lists the columns in the result set. Additional columns beyond column 5 (REMARKS) can be defined by the driver. An application should gain access to driver-specific columns by counting down from the end of the result set instead of specifying an explicit ordinal position. For more information, see Data Returned by Catalog Functions.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Column number</th>
<th>Data type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_CAT (ODBC 1.0)</td>
<td>1</td>
<td>Varchar</td>
<td>Catalog name; NULL if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;&quot;) for those tables that do not have catalogs.</td>
</tr>
<tr>
<td>TABLE_SCHEMA (ODBC 1.0)</td>
<td>2</td>
<td>Varchar</td>
<td>Schema name; NULL if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string (&quot;&quot;&quot;) for those tables that do not have schemas.</td>
</tr>
<tr>
<td>TABLE_NAME (ODBC 1.0)</td>
<td>3</td>
<td>Varchar</td>
<td>Table name.</td>
</tr>
<tr>
<td>TABLE_TYPE (ODBC 1.0)</td>
<td>4</td>
<td>Varchar</td>
<td>Table type name; one of the following: &quot;TABLE&quot;, &quot;VIEW&quot;, &quot;SYSTEM TABLE&quot;, &quot;GLOBAL TEMPORARY&quot;, &quot;LOCAL TEMPORARY&quot;, &quot;ALIAS&quot;, &quot;SYNONYM&quot;, or a data source–specific type name. The meanings of &quot;ALIAS&quot; and &quot;SYNONYM&quot; are driver-specific.</td>
</tr>
<tr>
<td>REMARKS (ODBC 1.0)</td>
<td>5</td>
<td>Varchar</td>
<td>A description of the table.</td>
</tr>
</tbody>
</table>

The following sample code does not free handles and connections. See SQLFreeHandle Function and SQLFreeStmt Function for code samples to free handles and statements.
// simple helper functions
int MySQLSuccess(SQLRETURN rc) {
    return (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO);
}

struct DataBinding {
    SQLSMALLINT TargetType;
    SQLPOINTER TargetValuePtr;
    SQLINTEGER BufferLength;
    SQLLEN StrLen_or_Ind;
};

void printCatalog(const struct DataBinding* catalogResult) {
    if (catalogResult[0].StrLen_or_Ind != SQL_NULL_DATA)
        printf("Catalog Name = %s\n", (char *)catalogResult[0].TargetValuePtr);
}

// remember to disconnect and free memory, and free statements and handles
int main() {
    int bufferSize = 1024, i, numCols = 5;
    struct DataBinding* catalogResult = (struct DataBinding*) malloc( numCols * sizeof(struct wchar_t*))
    wchar_t* dbName = (wchar_t*)malloc( sizeof(wchar_t)*bufferSize );
    wchar_t* userName = (wchar_t*)malloc( sizeof(wchar_t)*bufferSize );

    // declare and initialize the environment, connection, statement handles
    SQLHENV henv = NULL; // Environment
    SQLHDBC hdbc = NULL; // Connection handle
    SQLHSTMT hstmt = NULL; // Statement handle
    SQLRETURN retCode;
    HWND desktopHandle = GetDesktopWindow(); // desktop's window handle
    SQLWCHAR connStrbuffer[1024];
    SQLMALLINT connStrBufferLen;
    retCode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    retCode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, -1);
    retCode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    retCode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)10, 0);
    retCode = SQLDriverConnect(hdbc, desktopHandle, (SQLCHAR*)"Driver={SQL Server}", SQL_NT
    retCode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    retCode = SQLGetInfo(hdbc, SQL_DATABASE_NAME, dbName, (SQLMALLINT)bufferSize, (SQLSMALLINT)bufferSize);
    bufferSize = 1024;

    // allocate memory for the binding
    // free this memory when done
    for ( i = 0 ; i < numCols ; i++ ) {
        catalogResult[i].TargetType = SQL_C_CHAR;
        catalogResult[i].BufferLength = (bufferSize + 1);
        catalogResult[i].TargetValuePtr = malloc( sizeof(unsigned char)*catalogResult[i].Buf}
}
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding a buffer to a column in a result set</td>
<td>SQLBindCol Function</td>
</tr>
<tr>
<td>Canceling statement processing</td>
<td>SQLCancel Function</td>
</tr>
<tr>
<td>Returning privileges for a column or columns</td>
<td>SQLColumnPrivileges Function</td>
</tr>
<tr>
<td>Returning the columns in a table or tables</td>
<td>SQLColumns Function</td>
</tr>
<tr>
<td>Fetching a single row or a block of data in a forward-only direction</td>
<td>SQLFetch Function</td>
</tr>
<tr>
<td>Fetching a block of data or scrolling through a result set</td>
<td>SQLFetchScroll Function</td>
</tr>
<tr>
<td>Returning table statistics and indexes</td>
<td>SQLStatistics Function</td>
</tr>
<tr>
<td>Returning privileges for a table or tables</td>
<td>SQLTablePrivileges Function</td>
</tr>
</tbody>
</table>

See Also

ODBC API Reference
ODBC Header Files

© 2016 Microsoft

SQLTransact Function

Conformance
Version Introduced: ODBC 1.0 Standards Compliance: Deprecated

Summary
In ODBC 3.x, the ODBC 2.x function `SQLTransact` has been replaced by `SQLEndTran`. For more information, see `SQLEndTran`.

**Note**

The attribute `SQL_ASYNC_DBC_FUNCTION_ENABLE`, which was introduced in ODBC 3.8, is not supported by `SQLTransact`. Applications using an asynchronous operation on a connection handle must use `SQLEndTran`.

**See Also**

ODBC API Reference
ODBC Header Files

Setup DLL API Reference

This section describes the syntax of the driver setup DLL API, which consists of two functions (`ConfigDriver` and `ConfigDSN`). `ConfigDriver` and `ConfigDSN` can be either in the driver DLL or in a separate setup DLL.

In addition, this section describes the syntax of the translator setup DLL API, which consists of a single function (`ConfigTranslator`). `ConfigTranslator` can be either in the translator DLL or in a separate setup DLL.

Each function is labeled with the version of ODBC in which it was introduced.

This section contains the following topics.

- `ConfigDriver` Function
- `ConfigDSN` Function
- `ConfigTranslator` Function

**ConfigDriver Function**

**Conformance**

Version Introduced: ODBC 2.5

**Summary**

`ConfigDriver` allows a setup program to perform install and uninstall functions without requiring the program to call `ConfigDSN`. This function will perform driver-specific functions such as
creating driver-specific system information and performing DSN conversions during installation, as well as cleaning up system information modifications during uninstall. This function is exposed by the driver setup DLL or a separate setup DLL.

Syntax

```c
BOOL ConfigDriver( 
    HWND hwndParent,
    WORD fRequest,
    LPCSTR lpszDriver,
    LPCSTR lpszArgs,
    LPSTR lpszMsg,
    WORD cbMsgMax,
    WORD * pcbMsgOut);
```

Arguments

hwndParent
[Input] Parent window handle. The function will not display any dialog boxes if the handle is null.

fRequest
[Input] Type of request. The fRequest argument must contain one of the following values:

- ODBC_INSTALL_DRIVER: Install a new driver.
- ODBC_REMOVE_DRIVER: Remove a driver.

This option can also be driver-specific, in which case the fRequest argument for the first option must start from ODBC_CONFIG_DRIVER_MAX+1. The fRequest argument for any additional option must also start from a value greater than ODBC_CONFIG_DRIVER_MAX+1.

lpszDriver
[Input] The name of the driver as registered in the Odbcinst.ini key of the system information.

lpszArgs
[Input] A null-terminated string containing arguments for a driver-specific fRequest.

lpszMsg
[Output] A null-terminated string containing an output message from the driver setup.

cbMsgMax
[Input] Length of lpszMsg.

pcbMsgOut
[Output] Total number of bytes available to return in lpszMsg.

If the number of bytes available to return is greater than or equal to cbMsgMax, the output message in lpszMsg is truncated to cbMsgMax minus the null-termination character. The pcbMsgOut argument can be a null pointer.
Returns

The function returns TRUE if it is successful, FALSE if it fails.

Diagnostics

When `ConfigDriver` returns FALSE, an associated `*pfErrorCode` value is posted to the installer error buffer by a call to `SQLPostInstallerError` and can be obtained by calling `SQLInstallerError`. The following table lists the `*pfErrorCode` values that can be returned by `SQLInstallerError` and explains each one in the context of this function.

<table>
<thead>
<tr>
<th><code>*pfErrorCode</code></th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_INVALID_HWND</td>
<td>Invalid window handle</td>
<td>The <code>hwndParent</code> argument was invalid.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_REQUEST_TYPE</td>
<td>Invalid type of request</td>
<td>The <code>fRequest</code> argument was not one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ODBC_INSTALL_DRIVER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ODBC_REMOVE_DRIVER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The driver-specific option was less than or equal to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ODBC_CONFIG_DRIVER_MAX.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_NAME</td>
<td>Invalid driver or translator name</td>
<td>The <code>lpszDriver</code> argument was invalid. It could not be found in the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_REQUEST FAILED</td>
<td>Request failed</td>
<td>Could not perform the operation requested by the <code>fRequest</code> argument.</td>
</tr>
</tbody>
</table>
| ODBC_ERROR_DRIVER_SPECIFIC      | Driver- or translator-specific error | A driver-specific error for which there is no defined ODBC installer error. The `SzError` argument in a call to the `SQLPostInstallerError` function should contain the driver-specific error message.

Comments

Driver-Specific Options

An application can request driver-specific features exposed by the driver by using the `fRequest`
argument. The fRequest for the first option will be ODBC_CONFIG_DRIVER_MAX plus 1, and additional options will be incremented by 1 from that value. Any arguments required by the driver for that function should be provided in a null-terminated string passed in the lpszArgs argument. Drivers providing such functionality should maintain a table of driver-specific options. The options should be fully documented in driver documentation. Application writers who use driver-specific options should be aware that this will make the application less interoperable.

Messages

A driver setup routine can send a text message to an application as a null-terminated string in the lpszMsg buffer. The message will be truncated to cbMsgMax minus the null-termination character by the ConfigDriver function if it is greater than or equal to cbMsgMax characters.

ConfigDSN Function

Conformance

Version Introduced: ODBC 1.0

Summary

ConfigDSN adds, modifies, or deletes data sources from the system information. It may prompt the user for connection information. It can be in the driver DLL or a separate setup DLL.

Syntax

```c
BOOL ConfigDSN(
    HWND hwndParent,
    WORD fRequest,
    LPCSTR lpszDriver,
    LPCSTR lpszAttributes);
```

Arguments

hwndParent

[Input] Parent window handle. The function will not display any dialog boxes if the handle is null.

fRequest

[Input] Type of request. The fRequest argument must contain one of the following values:

- ODBC_ADD_DSN: Add a new data source.
ODBC_CONFIG_DSN: Configure (modify) an existing data source.

ODBC_REMOVE_DSN: Remove an existing data source.

*lpszDriver*

[Input] Driver description (usually the name of the associated DBMS) presented to users instead of the physical driver name.

*lpszAttributes*

[Input] A doubly null-terminated list of attributes in the form of keyword-value pairs. For more information, see "Comments."

Returns

The function returns TRUE if it is successful, FALSE if it fails.

Diagnostics

When ConfigDSN returns FALSE, an associated *pfErrorCode value is posted to the installer error buffer by a call to SQLPostInstallerError and can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_INVALID_HWND</td>
<td>Invalid window handle</td>
<td>The hwndParent argument was invalid.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_KEYWORD_VALUE</td>
<td>Invalid keyword-value pairs</td>
<td>The lpszAttributes argument contained a syntax error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_NAME</td>
<td>Invalid driver or translator name</td>
<td>The lpszDriver argument was invalid. It could not be found in the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_REQUEST_TYPE</td>
<td>Invalid type of request</td>
<td>The request argument was not one of the following: ODBC_ADD_DSN ODBC_CONFIG_DSN ODBC_REMOVE_DSN</td>
</tr>
<tr>
<td>ODBC_ERROR_REQUEST_FAILED</td>
<td>Request failed</td>
<td>Could not perform the operation requested by the fRequest argument.</td>
</tr>
<tr>
<td>ODBC_ERROR_DRIVER_SPECIFIC</td>
<td>Driver- or</td>
<td>A driver-specific error for which there is</td>
</tr>
</tbody>
</table>
translator-specific error  no defined ODBC installer error. The SzError argument in a call to the SQLPostInstallerError function should contain the driver-specific error message.

Comments

ConfigDSN receives connection information from the installer DLL as a list of attributes in the form of keyword-value pairs. Each pair is terminated with a null byte, and the entire list is terminated with a null byte. (That is, two null bytes mark the end of the list.) Spaces are not allowed around the equal sign in the keyword-value pair. ConfigDSN can accept keywords that are not valid keywords for SQLBrowseConnect and SQLDriverConnect. ConfigDSN does not necessarily support all keywords that are valid keywords for SQLBrowseConnect and SQLDriverConnect. (ConfigDSN does not accept the DRIVER keyword.) The keywords used by the ConfigDSN function must support all the options required to re-create the data source using the AUTO setup feature of the installer. When the uses of the ConfigDSN values and the connection string values are the same, the same keywords should be used.

As in SQLBrowseConnect and SQLDriverConnect, the keywords and their values should not contain the [{}()",;?*=!@ characters, and the value of the DSN keyword cannot consist only of blanks. Because of the registry grammar, keywords and data source names cannot contain the backslash (\) character.

ConfigDSN should call SQLValidDSN to check the length of the data source name and to verify that no invalid characters are included in the name. If the data source name is longer than SQL_MAX_DSN_LENGTH or includes invalid characters, SQLValidDSN returns an error and ConfigDSN returns an error. The length of the data source name is also checked by SQLWriteDSNToIni.

For example, to configure a data source that requires a user ID, password, and database name, a setup application might pass the following keyword-value pairs:

```
DSN=Personnel	Data\0UID=Smith\0PWD=Sesame\0DATABASE=Personnel\0\0
```

For more information about these keywords, see SQLDriverConnect and each driver's documentation.

To display a dialog box, hwndParent must not be null.

Adding a Data Source

If a data source name is passed to ConfigDSN in lpszAttributes, ConfigDSN checks that the name is valid. If the data source name matches an existing data source name and hwndParent is null, ConfigDSN overwrites the existing name. If it matches an existing name and hwndParent is not null, ConfigDSN prompts the user to overwrite the existing name.

If lpszAttributes contains enough information to connect to a data source, ConfigDSN can add the data source or display a dialog box with which the user can change the connection information. If lpszAttributes does not contain enough information to connect to a data source, ConfigDSN must
determine the necessary information; if hwndParent is not null, it displays a dialog box to retrieve the information from the user.

If ConfigDSN displays a dialog box, it must display any connection information passed to it in lpszAttributes. In particular, if a data source name was passed to it, ConfigDSN displays that name but does not allow the user to change it. ConfigDSN can supply default values for connection information not passed to it in lpszAttributes.

If ConfigDSN cannot get complete connection information for a data source, it returns FALSE.

If ConfigDSN can get complete connection information for a data source, it calls SQLWriteDSNToIni in the installer DLL to add the new data source specification to the Odbc.ini file (or registry). SQLWriteDSNToIni adds the data source name to the [ODBC Data Sources] section, creates the data source specification section, and adds the DRIVER keyword with the driver description as its value. ConfigDSN calls SQLWritePrivateProfileString in the installer DLL to add any additional keywords and values used by the driver.

Modifying a Data Source

To modify a data source, a data source name must be passed to ConfigDSN in lpszAttributes. ConfigDSN checks that the data source name is in the Odbc.ini file (or registry).

If hwndParent is null, ConfigDSN uses the information in lpszAttributes to modify the information in the Odbc.ini file (or registry). If hwndParent is not null, ConfigDSN displays a dialog box using the information in lpszAttributes; for information not in lpszAttributes, it uses information from the system information. The user can modify the information before ConfigDSN stores it in the system information.

If the data source name was changed, ConfigDSN first calls SQLRemoveDSNFromIni in the installer DLL to remove the existing data source specification from the Odbc.ini file (or registry). It then follows the steps in the preceding section to add the new data source specification. If the data source name was not changed, ConfigDSN calls SQLWritePrivateProfileString in the installer DLL to make any other changes. ConfigDSN may not delete or change the value of the Driver keyword.

Deleting a Data Source

To delete a data source, a data source name must be passed to ConfigDSN in lpszAttributes. ConfigDSN checks that the data source name is in the Odbc.ini file (or registry). It then calls SQLRemoveDSNFromIni in the installer DLL to remove the data source.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding, modifying, or removing a data source</td>
<td>SQLConfigDataSource</td>
</tr>
<tr>
<td>Getting a value from the Odbc.ini file or the registry</td>
<td>SQLGetPrivateProfileString</td>
</tr>
<tr>
<td>Removing the default data source</td>
<td>SQLRemoveDefaultDataSource</td>
</tr>
</tbody>
</table>
ConfigTranslator Function

Conformance
Version Introduced: ODBC 2.0

Summary
ConfigTranslator returns a default translation option for a translator. It can be in the translator DLL or a separate setup DLL.

Syntax

```c
BOOL ConfigTranslator(
    HWND hwndParent,
    DWORD * pvOption);
```

Arguments

hwndParent

[Input] Parent window handle. The function will not display any dialog boxes if the handle is null.

pvOption


Returns

The function returns TRUE if it is successful, FALSE if it fails.

Diagnostics
When `ConfigTranslator` returns FALSE, an associated `*pfErrorCode` value is posted to the installer error buffer by a call to `SQLPostInstallerError` and can be obtained by calling `SQLInstallerError`. The following table lists the `*pfErrorCode` values that can be returned by `SQLInstallerError` and explains each one in the context of this function.

<table>
<thead>
<tr>
<th><code>*pfErrorCode</code></th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_INVALID_HWND</td>
<td>Invalid window handle</td>
<td>The <code>hwndParent</code> argument was invalid or NULL.</td>
</tr>
<tr>
<td>ODBC_ERROR_DRIVER_SPECIFIC</td>
<td>Driver- or translator-specific error</td>
<td>A driver-specific error for which there is no defined ODBC installer error. The <code>SzError</code> argument in a call to the <code>SQLPostInstallerError</code> function should contain the driver-specific error message.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_OPTION</td>
<td>Invalid translation option</td>
<td>The <code>pvOption</code> argument contained an invalid value.</td>
</tr>
</tbody>
</table>

**Comments**

If the translator supports only a single translation option, `ConfigTranslator` returns TRUE and sets `pvOption` to the 32-bit option. Otherwise, it determines the default translation option to use. `ConfigTranslator` can display a dialog box with which a user selects a default translation option.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting a translation option</td>
<td>SQLGetConnectAttr</td>
</tr>
<tr>
<td>Selecting a translator</td>
<td>SQLGetTranslator</td>
</tr>
<tr>
<td>Setting a translation option</td>
<td>SQLSetConnectAttr</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**Installer DLL API Reference Function**

This section describes the syntax of the functions in the installer DLL API. The installer DLL API consists
of 20 functions. Three of these functions, SQLGetTranslator, SQLRemoveDSNFromIni, and SQLWriteDSNToIni, are called only by setup DLLs. The other functions are called by the setup and administration programs.

Each function is labeled with the version of ODBC in which it was introduced.

This section contains the following topics.

- SQLConfigDataSource Function
- SQLConfigDriver Function
- SQLCreateDataSource Function
- SQLGetConfigMode Function
- SQLGetInstalledDrivers Function
- SQLGetPrivateProfileString Function
- SQLGetTranslator Function
- SQLInstallDriverEx Function
- SQLInstallDriverManager Function
- SQLInstallerError Function
- SQLInstallTranslator Function
- SQLInstallTranslatorEx Function
- SQLManageDataSources Function
- SQLPostInstallerError Function
- SQLReadFileDSN Function
- SQLRemoveDefaultDataSource Function
- SQLRemoveDriver Function
- SQLRemoveDriverManager Function
- SQLRemoveDSNFromIni Function
- SQLRemoveTranslator Function
- SQLSetConfigMode Function
- SQLValidDSN Function
- SQLWriteDSNToIni Function
- SQLWriteFileDSN Function
- SQLWritePrivateProfileString Function
SQLConfigDataSource Function

Conformance
Version Introduced: ODBC 1.0

Summary
SQLConfigDataSource adds, modifies, or deletes data sources.

The functionality of SQLConfigDataSource can also be accessed with ODBCCONF.EXE.

Syntax

```c
BOOL SQLConfigDataSource(
    HWND hwndParent,
    WORD fRequest,
    LPCSTR lpszDriver,
    LPCSTR lpszAttributes);
```

Arguments

hwndParent
[Input] Parent window handle. The function will not display any dialog boxes if the handle is null.

fRequest
[Input] Type of request. The fRequest argument must contain one of the following values:

ODBC_ADD_DSN: Add a new user data source.

ODBC_CONFIG_DSN: Configure (modify) an existing user data source.

ODBC_REMOVE_DSN: Remove an existing user data source.

ODBC_ADD_SYS_DSN: Add a new system data source.

ODBC_CONFIG_SYS_DSN: Modify an existing system data source.

ODBC_REMOVE_SYS_DSN: Remove an existing system data source.

ODBC_REMOVE_DEFAULT_DSN: Remove the default data source specification section from the system information. (It also removes the default driver specification section from the Odbcinst.ini entry in the system information. This fRequest performs the same function as the deprecated SQLRemoveDefaultDataSource function.) When this option is specified, all of the other parameters in the call to SQLConfigDataSource should be NULLs; if they are not NULL, they will be ignored.

lpszDriver
[Input] Driver description (usually the name of the associated DBMS) presented to users instead
of the physical driver name.

**lpszAttributes**

[Input] A doubly null-terminated list of attributes in the form of keyword-value pairs. For more information, see **ConfigDSN**.

**Returns**

The function returns TRUE if it is successful, FALSE if it fails. If no entry exists in the system information when this function is called, the function returns FALSE.

**Diagnostics**

When **SQLConfigDataSource** returns FALSE, an associated *pfErrorCode* value can be obtained by calling **SQLInstallerError**. The following table lists the *pfErrorCode* values that can be returned by **SQLInstallerError** and explains each one in the context of this function.

<table>
<thead>
<tr>
<th><em>pfErrorCode</em></th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_HWND</td>
<td>Invalid window handle</td>
<td>The hwndParent argument was invalid or NULL.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_REQUEST_TYPE</td>
<td>Invalid type of request</td>
<td>The fRequest argument was not one of the following: ODBC_ADD_DSN ODBC_CONFIG_DSN ODBC_REMOVE_DSN ODBC_ADD_SYS_DSN ODBC_CONFIG_SYS_DSN ODBC_REMOVE_SYS_DSN ODBC_REMOVE_DEFAULT_DSN</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_NAME</td>
<td>Invalid driver or translator name</td>
<td>The lpszDriver argument was invalid. It could not be found in the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_KEYWORD_VALUE</td>
<td>Invalid keyword-value pairs</td>
<td>The lpszAttributes argument contained a syntax error.</td>
</tr>
<tr>
<td>ODBC_ERROR_REQUEST_FAILED</td>
<td>Request failed</td>
<td>The installer could not perform the operation requested by the fRequest argument. The call to <strong>ConfigDSN</strong> failed.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>ODBC_ERROR_LOAD_LIBRARY_FAILED</td>
<td>Could not load the driver or translator setup library</td>
<td></td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td></td>
</tr>
</tbody>
</table>

**Comments**

*SQLConfigDataSource* uses the value of *lpzDriver* to read the full path of the setup DLL for the driver from the system information. It loads the DLL and calls *ConfigDSN* with the same arguments that were passed to it.

*SQLConfigDataSource* returns FALSE if it is unable to find or load the setup DLL or if the user cancels the dialog box. Otherwise, it returns the status it received from *ConfigDSN*.

*SQLConfigDataSource* maps the System DSN *fRequest* to the User DSN *fRequest* (ODBC_ADD_SYS_DSN to ODBC_ADD_DSN, ODBC_CONFIG_SYS_DSN to ODBC_CONFIG_DSN, and ODBC_REMOVE_SYS_DSN to ODBC_REMOVE_DSN). To distinguish user and System DSNs, *SQLConfigDataSource* sets the installer configuration mode according to the following table. Prior to returning, *SQLConfigDataSource* resets configuration mode to BOTHDSN. *ConfigDSN* (implemented by drivers) should call *SQLWriteDSNToIni* and *SQLWritePrivateProfileString* to support a system DSN. For more information, see *ConfigDSN Function*.

<table>
<thead>
<tr>
<th><em>fRequest</em></th>
<th>Configuration mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ADD_DSN</td>
<td>USERDSN_ONLY</td>
</tr>
<tr>
<td>ODBC_CONFIG_DSN</td>
<td>USERDSN_ONLY</td>
</tr>
<tr>
<td>ODBC_REMOVE_DSN</td>
<td>USERDSN_ONLY</td>
</tr>
<tr>
<td>ODBC_ADD_SYS_DSN</td>
<td>SYSTEMDSN_ONLY</td>
</tr>
<tr>
<td>ODBC_CONFIG_SYS_DSN</td>
<td>SYSTEMDSN_ONLY</td>
</tr>
<tr>
<td>ODBC_REMOVE_SYS_DSN</td>
<td>SYSTEMDSN_ONLY</td>
</tr>
</tbody>
</table>

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Adding, modifying, or removing a data source: ConfigDSN (in the setup DLL)

Removing a data source name from the system information: SQLRemoveDSNFromIni

Adding a data source name to the system information: SQLWriteDSNToIni

---

© 2016 Microsoft

# SQLConfigDriver Function

**Conformance**
Version Introduced: ODBC 2.5

**Summary**
SQLConfigDriver loads the appropriate driver setup DLL and calls the ConfigDriver function.

The functionality of SQLConfigDriver can also be accessed with ODBCCONF.EXE.

**Syntax**

```c
BOOL SQLConfigDriver(
    HWND hwndParent,
    WORD fRequest,
    LPCSTR lpszDriver,
    LPCSTR lpszArgs,
    LPSTR lpszMsg,
    WORD cbMsgMax,
    WORD * pcbMsgOut);
```

**Arguments**

- **hwndParent**
  [Input] Parent window handle. The function will not display any dialog boxes if the handle is null.

- **fRequest**
  [Input] Type of request. fRequest must contain one of the following values:

  - ODBC_CONFIG_DRIVER: Changes the connection pooling timeout used by the driver.
  - ODBC_INSTALL_DRIVER: Installs a new driver.
  - ODBC_REMOVE_DRIVER: Removes an existing driver.

  This option can also be driver-specific, in which case the fRequest for the first option must start
from ODBC_CONFIG_DRIVER_MAX+1. The \texttt{fRequest} for any additional option must also start from a value greater than ODBC_CONFIG_DRIVER_MAX+1.

\textit{lpszDriver}

[Input] The name of the driver as registered in the system information.

\textit{lpszArgs}

[Input] A null-terminated string that contains arguments for a driver-specific \texttt{fRequest}.

\textit{lpszMsg}

[Output] A null-terminated string that contains an output message from the driver setup.

\textit{cbMsgMax}

[Input] Length of \textit{lpszMsg}.

\textit{pcbMsgOut}

[Output] Total number of bytes available to return in \textit{lpszMsg}. If the number of bytes available to return is greater than or equal to \textit{cbMsgMax}, the output message in \textit{lpszMsg} is truncated to \textit{cbMsgMax} minus the null-termination character. The \textit{pcbMsgOut} argument can be a null pointer.

\section*{Returns}

The function returns TRUE if it is successful, FALSE if it fails.

\section*{Diagnostics}

When \texttt{SQLConfigDriver} returns FALSE, an associated \*\texttt{pfErrorCode} value can be obtained by calling \texttt{SQLInstallerError}. The following table lists the \*\texttt{pfErrorCode} values that can be returned by \texttt{SQLInstallerError} and explains each one in the context of this function.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{*pfErrorCode} & \textbf{Error} & \textbf{Description} \\
\hline
ODBC_ERROR_GENERAL_ERR & General installer error & An error occurred for which there was no specific installer error. \\
\hline
ODBC_ERROR_INVALID_BUFF_LEN & Invalid buffer length & The \textit{lpszMsg} argument was invalid. \\
\hline
ODBC_ERROR_INVALID_HWND & Invalid window handle & The \textit{hwndParent} argument was invalid. \\
\hline
ODBC_ERROR_INVALID_REQUEST_TYPE & Invalid type of request & The \textit{fRequest} argument was not one of the following: ODBC_INSTALL_DRIVER ODBC_REMOVE_DRIVER The \textit{fRequest} argument was a driver-specific option that was less than or equal to \\
\hline
\end{tabular}
\end{table}
ODBC_ERROR_INVALID_NAME
Invalid driver or translator name
The lpszDriver argument was invalid. It could not be found in the registry.

ODBC_ERROR_INVALID_KEYWORD_VALUE
Invalid keyword-value pairs
The lpszArgs argument contained a syntax error.

ODBC_ERROR_REQUEST_FAILED
Request failed
The installer could not perform the operation requested by the fRequest argument. The call to ConfigDriver failed.

ODBC_ERROR_LOAD_LIBRARY_FAILED
Could not load the driver or translator setup library
The driver setup library could not be loaded.

ODBC_ERROR_OUT_OF_MEM
Out of memory
The installer could not perform the function because of a lack of memory.

Comments

SQLConfigDriver allows an application to call a driver's ConfigDriver routine without having to know the name and load the driver-specific setup DLL. A Setup program calls this function after the driver setup DLL has been installed. The calling program should be aware that this function might not be available for all drivers. In such a case, the calling program should continue without error.

Driver-Specific Options

An application can request driver-specific features exposed by the driver by using the fRequest argument. The fRequest for the first option will be ODBC_CONFIG_DRIVER_MAX+1, and additional options will be incremented by 1 from that value. Any arguments required by the driver for that function should be provided in a null-terminated string passed in the lpszArgs argument. Drivers providing such functionality should maintain a table of driver-specific options. The options should be fully documented in driver documentation. Application writers who use driver-specific options should be aware that this use will make the application less interoperable.

Setting Connection Pooling Timeout

Connection pooling timeout properties can be set when you set the configuration of the driver. SQLConfigDriver is called with an fRequest of ODBC_CONFIG_DRIVER and lpszArgs set to CPTimeout. CPTimeout determines the period of time that a connection can remain in the connection pool without being used. When the timeout expires, the connection is closed and removed from the pool. The default timeout is 60 seconds.
When `SQLConfigDriver` is called with `fRequest` set to `ODBC_INSTALL_DRIVER` or `ODBC_REMOVE_DRIVER`, the Driver Manager loads the appropriate driver setup DLL and calls the `ConfigDriver` function. When `SQLConfigDriver` is called with an `fRequest` of `ODBC_CONFIG_DRIVER`, all processing is performed in the ODBC installer, so that the driver setup DLL does not have to be loaded.

**Messages**

A driver setup routine can send a text message to an application as null-terminated strings in the `lpszMsg` buffer. The message will be truncated to `cbMsgMax` minus the null-termination character by the `ConfigDriver` function if it is greater than or equal to `cbMsgMax` characters.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding, modifying, or removing a driver</td>
<td><code>ConfigDriver</code> (in the setup DLL)</td>
</tr>
<tr>
<td>Removing the default data source</td>
<td><code>SQLRemoveDefaultDataSource</code></td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**SQLCreateDataSource Function**

**Conformance**

Version Introduced: ODBC 2.0

**Summary**

`SQLCreateDataSource` displays a dialog box with which the user can add a data source.

**Syntax**

```c
BOOL SQLCreateDataSource(
    HWND hwnd,
    LPSTR lpszDS);
```

**Arguments**
hwnd
[Input] Parent window handle.

lpszDS
[Input] Data source name. lpszDS can be a null pointer or an empty string.

Returns

SQLCreateDataSource returns TRUE if the data source is created. Otherwise, it returns FALSE.

Diagnostics

When SQLCreateDataSource returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_HWND</td>
<td>Invalid window handle</td>
<td>The hwnd argument was invalid or NULL.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_DSN</td>
<td>Invalid DSN</td>
<td>The lpszDS argument contained a string that was invalid for a DSN.</td>
</tr>
<tr>
<td>ODBC_ERROR_REQUEST_FAILED</td>
<td>Request failed</td>
<td>The call to ConfigDSN with the ODBC_ADD_DSN option failed.</td>
</tr>
<tr>
<td>ODBC_ERROR_LOAD_LIBRARY_FAILED</td>
<td>Could not load the driver or</td>
<td>The driver setup library could not be loaded.</td>
</tr>
<tr>
<td></td>
<td>translator setup library</td>
<td></td>
</tr>
<tr>
<td>ODBC_ERROR_USER_CANCELED</td>
<td>User canceled operation</td>
<td>User canceled creation of a new data source.</td>
</tr>
<tr>
<td>ODBC_ERROR_CREATE_DSN_FAILED</td>
<td>Could not create the requested DSN</td>
<td>Could not connect to the database; the call to SQLDriverConnect for a File DSN did not return a successful connection. Could not write to the file.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>
Comments

If hwnd is null, SQLCreateDataSource returns FALSE. Otherwise, it displays the Create New Data Source dialog box with a wizard page for choosing the type of data source to be set up, as shown in the following illustration.

The default option is File Data Source. When a data source has been chosen and Next clicked, the following wizard page that contains a list of installed drivers is displayed.

If Cancel is clicked, the dialog box disappears and SQLCreateDataSource returns FALSE with the error code of ODBC_ERROR_USER_CANCELED. If either the User Data Source or System Data Source option was selected, the Advanced button is unavailable.

When the Next button is clicked, one of the following will occur, depending on which type of data source was selected:
If **File Data Source** was selected, a wizard page is displayed for the user to enter a file name.

If either **User Data Source** or **System Data Source** was selected, a wizard page displaying the type of data source and driver is displayed for review, and when **Finish** is clicked, the data source is set up.

If **Advanced** is clicked from the Create New Data Source wizard page, a wizard page is displayed for the user to enter driver-specific information. In the text box of this dialog box, type the driver and keywords separated by returns, as shown in the following illustration.

```
Server=MyServer
Database=MyDatabase
```

Additional driver-specific keywords can be found under the description of **SQLDriverConnect**. All except **DSN** are allowed.

The default for the **Verify This Connection** option is TRUE. This default applies whether or not this wizard page is activated. If **OK** is clicked, the string specified in the text box and the **Verify This Connection** option value are cached. (If the **Close** button or **Cancel** is clicked, any newly entered driver-specific information is lost because the string specified in the text box and the **Verify This Connection** option value are not cached.)

If **File Data Source** was selected in the first wizard page, then after a driver has been selected and the keyword values have been entered in the Advanced wizard page, the user is prompted to enter a file name. Click **Browse** to search for a file name, in which case the default directory in the **Browse** box is specified by a combination of the path specified by `CommonFileDir` in `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion` and "ODBC\DataSources". (If `CommonFileDir` was "C:\Program Files\Common Files", the default directory would be "C:\Program Files\Common Files\ODBC\DataSources".)

When a file name has been entered and **Next** is clicked, the file name entered is checked for validity against the standard file-naming rules of the operating system. If the file name is invalid, an error message box notifies the user that an invalid file name was entered. After the user acknowledges the message box, the focus is returned to the wizard page in which the file name is entered. If the file name is valid, a wizard page that shows the selected keyword-value pairs is displayed for review, as shown in the following illustration.
If Finish is clicked and File Data Source was selected as the data source type, and if the Verify this connection option is TRUE, SQLDriverConnect is called with the SAVEFILE and DRIVER keywords. The DriverCompletion argument is set to SQL_DRIVER_COMPLETE. The file name for the SAVEFILE keyword is the name that was entered or chosen, and the driver name for the DRIVER keyword is the name that was chosen. If a driver-specific connection string was specified in the Advanced wizard page, that string is appended after the DRIVER keyword.

If SQLDriverConnect returns SQL_SUCCESS, the Driver Manager has created the File DSN. SQLCreateDataSource returns TRUE. If SQLDriverConnect does not return SQL_SUCCESS, a warning message box indicates that a connection could not be made to the data source. A DSN with minimal connection information can still be created. This message box lets the user either cancel or continue with the File DSN creation.

If the user chooses to continue creating the DSN, this process continues as if the Verify this connection option were set to FALSE. If the user chooses to cancel, FALSE is returned for SQLCreateDataSource with an error code of ODBC_ERROR_CREATE_DSN_FAILED.

If File Data Source was selected as the data source type and the Verify this connection option is FALSE, a File DSN is created with the DRIVER keyword and user-specified connect string (if any) from the Advanced wizard page. If the file creation was successful, TRUE is returned for SQLCreateDataSource. If the file creation was not successful, an error message box notifies the user with whatever error was returned from the operating system. FALSE is returned for SQLCreateDataSource with an error code of ODBC_ERROR_CREATE_DSN_FAILED. For more information about file data sources, see Connecting Using File Data Sources, or see SQLDriverConnect.

If User or System Data Source was selected as the data source type, ConfigDSN in the driver setup library is called with the ODBC_ADD_DSN fRequest. For more information, see ConfigDSN.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Managing data sources</td>
<td>SQLManageDataSources</td>
</tr>
</tbody>
</table>
**SQLGetConfigMode Function**

**Conformance**
Version Introduced: ODBC 3.0

**Summary**
*SQLGetConfigMode* retrieves the configuration mode that indicates where the Odbc.ini entry listing DSN values is in the system information.

**Syntax**

```c
BOOL SQLGetConfigMode(
    UWORD * pwConfigMode);
```

**Arguments**

*pwConfigMode*
[Output] Pointer to the buffer containing the configuration mode. (See "Comments.") The value in *pwConfigMode* can be:

- ODBC_USER_DSN
- ODBC_SYSTEM_DSN
- ODBC_BOTH_DSN

**Returns**
The function returns TRUE if it is successful, FALSE if it fails.

**Diagnostics**
When *SQLGetConfigMode* returns FALSE, an associated *pfErrorCode* value can be obtained by calling *SQLInstallerError*. The following table lists the *pfErrorCode* values that can be returned by *SQLInstallerError* and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comments

This function is used to determine where the Odbc.ini entry listing DSN values is in the system information. If *pwConfigMode is ODBC_USER_DSN, the DSN is a User DSN and the function reads from the Odbc.ini entry in HKEY_CURRENT_USER. If it is ODBC_SYSTEM_DSN, the DSN is a System DSN and the function reads from the Odbc.ini entry in HKEY_LOCAL_MACHINE. If it is ODBC_BOTH_DSN, HKEY_CURRENT_USER is tried, and if it fails, HKEY_LOCAL_MACHINE is used.

By default, SQLGetConfigMode returns ODBC_BOTH_DSN. When a User DSN or a System DSN is created by a call to SQLConfigDataSource, the function sets the configuration mode to ODBC_USER_DSN or ODBC_SYSTEM_DSN to distinguish user and System DSNs while modifying a DSN. Prior to returning, SQLConfigDataSource resets the configuration mode to ODBC_BOTH_DSN.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting the configuration mode</td>
<td>SQLSetConfigMode</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

SQLGetInstalledDrivers Function

Conformance

Version Introduced: ODBC 1.0

Summary

SQLGetInstalledDrivers reads the [ODBC Drivers] section of the system information and returns a list of descriptions of the installed drivers.

Syntax

```c
BOOL SQLGetInstalledDrivers(
LPSTR lpszBuf,
WORD cbBufMax,
WORD * pcbBufOut);
```
Arguments

`lpszBuf`

[Output] List of descriptions of the installed drivers. For information about the list structure, see "Comments."

`cbBufMax`

[Input] Length of `lpszBuf`.

`pcbBufOut`

[Output] Total number of bytes (excluding the null-termination byte) returned in `lpszBuf`. If the number of bytes available to return is greater than or equal to `cbBufMax`, the list of driver descriptions in `lpszBuf` is truncated to `cbBufMax` minus the null-termination character. The `pcbBufOut` argument can be a null pointer.

Returns

The function returns TRUE if it is successful, FALSE if it fails.

Diagnostics

When `SQLGetInstalledDrivers` returns FALSE, an associated `*pfErrorCode` value can be obtained by calling `SQLInstallerError`. The following table lists the `*pfErrorCode` values that can be returned by `SQLInstallerError` and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_BUFF_LEN</td>
<td>Invalid buffer length</td>
<td>The <code>lpszBuf</code> argument was NULL or invalid, or the <code>cbBufMax</code> argument was less than or equal to 0.</td>
</tr>
<tr>
<td>ODBC_ERROR_COMPONENT_NOT_FOUND</td>
<td>Component not found in registry</td>
<td>The installer could not find the [ODBC Drivers] section in the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

Comments
Each driver description is terminated with a null byte, and the entire list is terminated with a null byte. (That is, two null bytes mark the end of the list.) If the allocated buffer is not large enough to hold the entire list, the list is truncated without error. An error is returned if a null pointer is passed in as lpszBuf.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returning driver descriptions and attributes</td>
<td>SQLDrivers</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

### SQLGetPrivateProfileString Function

**Conformance**
Version Introduced: ODBC 2.0

**Summary**
SQLGetPrivateProfileString gets a list of names of values or data corresponding to a value of the system information.

**Syntax**

```c
int SQLGetPrivateProfileString(
    LPCSTR lpszSection,
    LPCSTR lpszEntry,
    LPCSTR lpszDefault,
    LPCSTR RetBuffer,
    INT cbRetBuffer,
    LPCSTR lpszFilename);
```

**Arguments**

**lpszSection**
[Input] Points to a null-terminated string that specifies the section containing the key name. If this argument is NULL, the function copies all section names in the file to the supplied buffer.

**lpszEntry**
[Input] Points to the null-terminated string containing the key name whose associated string is to be retrieved. If this argument is NULL, all key names in the section specified by the lpszSection argument are copied to the buffer specified by the RetBuffer argument.
**Returns**

SQLGetPrivateProfileString returns an integer value that indicates the number of characters read.

**Diagnostics**

When a call to SQLGetPrivateProfileString fails, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

**Comments**

SQLGetPrivateProfileString is provided as a simple way to port drivers and driver setup DLLs from Microsoft® Windows® to Microsoft Windows NT®/Windows 2000. Calls to GetPrivateProfileString that retrieve a profile string from the Odbc.ini file should be replaced with calls to SQLGetPrivateProfileString. SQLGetPrivateProfileString calls functions in the Win32® API to retrieve the requested names of values or data corresponding to a value of the Odbc.ini subkey of the system information.

The configuration mode (as set by SQLSetConfigMode) indicates where the Odbc.ini entry listing DSN values is in the system information. If the DSN is a User DSN (the configuration mode is USERDSN_ONLY), the function reads from the Odbc.ini entry in HKEY_CURRENT_USER. If the DSN is a System DSN (SYSTEMDSN_ONLY), the function reads from the Odbc.ini entry in HKEY_LOCAL_MACHINE. If the configuration mode is BOTHDSN, HKEY_CURRENT_USER is tried, and if
it fails, HKEY_LOCAL_MACHINE is used.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing a value to the system information</td>
<td>SQLWritePrivateProfileString</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

SQLGetTranslator Function

Conformance

Version Introduced: ODBC 2.0

Summary

SQLGetTranslator displays a dialog box from which a user can select a translator.

Syntax

```c
BOOL SQLGetTranslator(
    HWND hwndParent,
    LPSTR lpszName,
    WORD cbNameMax,
    WORD * pcbNameOut,
    LPSTR lpszPath,
    WORD cbPathMax,
    WORD * pcbPathOut,
    DWORD * pvOption);
```

Arguments

hwndParent
  [Input] Parent window handle.

lpszName
  [Input/Output] Name of the translator from the system information.

cbNameMax
  [Input] Maximum length of the lpszName buffer.

pcbNameOut
Total number of bytes (excluding the null-termination byte) passed or returned in *lpszName*. If the number of bytes available to return is greater than or equal to *cbNameMax*, the translator name in *lpszName* is truncated to *cbNameMax* minus the null-termination character. The *pcbNameOut* argument can be a null pointer.

**lpszPath**
[Output] Full path of the translation DLL.

**cbPathMax**
[Input] Maximum length of the *lpszPath* buffer.

**pcbPathOut**
[Output] Total number of bytes (excluding the null-termination byte) returned in *lpszPath*. If the number of bytes available to return is greater than or equal to *cbPathMax*, the translation DLL path in *lpszPath* is truncated to *cbPathMax* minus the null-termination character. The *pcbPathOut* argument can be a null pointer.

**pvOption**

**Returns**

The function returns TRUE if it is successful, FALSE if it fails or if the user cancels the dialog box.

**Diagnostics**

When SQLGetTranslator returns FALSE, an associated *pfErrorCode* value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode* values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th><em>pfErrorCode</em></th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_BUFF_LEN</td>
<td>Invalid buffer length</td>
<td>The <em>cbNameMax</em> or <em>cbPathMax</em> argument was less than or equal to 0.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_HWND</td>
<td>Invalid window handle</td>
<td>The <em>hwndParent</em> argument was invalid or NULL.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_NAME</td>
<td>Invalid driver or translator name</td>
<td>The <em>lpszName</em> argument was invalid. It could not be found in the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_LOAD_LIBRARY_FAILED</td>
<td>Could not load the driver or translator setup library</td>
<td>The translator library could not be loaded.</td>
</tr>
</tbody>
</table>
Invalid transaction option
The pvOption argument contained an invalid value.

Out of memory
The installer could not perform the function because of a lack of memory.

Comments

If hwndParent is null or if lpszName, lpszPath, or pvOption is a null pointer, SQLGetTranslator returns FALSE. Otherwise, it displays the list of installed translators in the following dialog box.

If lpszName contains a valid translator name, it is selected. Otherwise, <No Translator> is selected.

If the user chooses <No Translator>, the contents of lpszName, lpszPath, and pvOption are not touched. SQLGetTranslator sets pcbNameOut and pcbPathOut to 0 and returns TRUE.

If the user chooses a translator, SQLGetTranslator calls ConfigTranslator in the translator's setup DLL. If ConfigTranslator returns FALSE, SQLGetTranslator returns to its dialog box. If ConfigTranslator returns TRUE, SQLGetTranslator returns TRUE, along with the selected translator name, path, and translation option.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuring a translator</td>
<td>ConfigTranslator</td>
</tr>
<tr>
<td>Getting a translation attribute</td>
<td>SQLGetConnectAttr</td>
</tr>
<tr>
<td>Setting a translation attribute</td>
<td>SQLSetConnectAttr</td>
</tr>
</tbody>
</table>
SQLInstallDriverEx Function

Conformance
Version Introduced: ODBC 3.0

Summary
SQLInstallDriverEx adds information about the driver to the Odbcinst.ini entry in the system information and increments the driver’s UsageCount by 1. However, if a version of the driver already exists but the UsageCount value for the driver does not exist, the new UsageCount value is set to 2.

This function does not actually copy any files. It is the responsibility of the calling program to copy the driver's files to the target directory properly.

The functionality of SQLInstallDriverEx can also be accessed with ODBCCONF.EXE.

Syntax

```c
BOOL SQLInstallDriverEx(
    LPCSTR lpszDriver,
    LPCSTR lpszPathIn,
    LPSTR lpszPathOut,
    WORD cbPathOutMax,
    WORD * pcbPathOut,
    WORD fRequest,
    LPDWORD lpdwUsageCount);
```

Arguments

*lpszDriver*  
[Input] The driver description (usually the name of the associated DBMS) presented to users instead of the physical driver name. The *lpszDriver* argument must contain a doubly null-terminated list of keyword-value pairs describing the driver. For more information about keyword-value pairs, see Driver Specification Subkeys. For more information about the doubly null-terminated string, see ConfigDSN Function.

*lpszPathIn*  
[Input] Full path of the target directory of the installation, or a null pointer. If *lpszPathIn* is a null pointer, the drivers will be installed in the system directory.

*lpszPathOut*  
[Output] Path of the target directory where the driver should be installed. If the driver has not previously been installed, *lpszPathOut* should be the same as *lpszPathIn*. If the driver was previously installed, *lpszPathOut* is the path of the previous installation.
cbPathOutMax

[Input] Length of lpszPathOut.

pcbPathOut

[Output] Total number of bytes (excluding the null-termination character) available to return in lpszPathOut. If the number of bytes available to return is greater than or equal to cbPathOutMax, the output path in lpszPathOut is truncated to cbPathOutMax minus the null-termination character. The pcbPathOut argument can be a null pointer.

fRequest

[Input] Type of request. The fRequest argument must contain one of the following values:

ODBC_INSTALL_INQUIRY: Inquire about where a driver can be installed.

ODBC_INSTALL_COMPLETE: Complete the installation request.

lpdwUsageCount

[Output] The usage count of the driver after this function has been called. Applications should not set the usage count. ODBC will maintain this count.

Returns

The function returns TRUE if it is successful, FALSE if it fails.

Diagnostics

When SQLInstallDriverEx returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_BUFF_LEN</td>
<td>Invalid buffer length</td>
<td>The lpszPathOut argument was not large enough to contain the output path. The buffer contains the truncated path. The cbPathOutMax argument was 0, and fRequest was ODBC_INSTALL_COMPLETE.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_REQUEST_TYPE</td>
<td>Invalid type of request</td>
<td>The fRequest argument was not one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ODBC_INSTALL_INQUIRY ODBC_INSTALL_COMPLETE</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_KEYWORD_VALUE</td>
<td>Invalid keyword-value pairs</td>
<td>The \textit{lpszDriver} argument contained a syntax error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_PATH</td>
<td>Invalid install path</td>
<td>The \textit{lpszPathIn} argument contained an invalid path.</td>
</tr>
<tr>
<td>ODBC_ERROR_LOAD_LIBRARY_FAILED</td>
<td>Could not load the driver or translator setup library</td>
<td>The driver setup library could not be loaded.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_PARAM_SEQUENCE</td>
<td>Invalid parameter sequence</td>
<td>The \textit{lpszDriver} argument did not contain a list of keyword-value pairs.</td>
</tr>
<tr>
<td>ODBC_ERROR_USAGE_UPDATE_FAILED</td>
<td>Could not increment or decrement the component usage count</td>
<td>The installer failed to increment the driver's usage count.</td>
</tr>
</tbody>
</table>

**Comments**

The \textit{lpszDriver} argument is a list of attributes in the form of keyword-value pairs. Each pair is terminated with a null byte, and the entire list is terminated with a null byte. (That is, two null bytes mark the end of the list.) The format of this list is as follows:

\begin{verbatim}
driver-desc\0Driver=driver-DLL-filename\0[Setup=setup-DLL-filename]\0
[driver-attr-keyword1=value1]\0[driver-attr-keyword2=value2]\0...
\0
\end{verbatim}

where \textbackslash 0 is a null byte and \textit{driver-attr-keywordn} is any driver attribute keyword. The keywords must appear in the specified order. For example, suppose a driver for formatted text files has separate driver and setup DLLs and can use files with the .txt and .csv extensions. The \textit{lpszDriver} argument for this driver might be as follows:

\begin{verbatim}
Text\0Driver=TEXT.DLL\0Setup=TXTSETUP.DLL\0FileUsage=1\0
FileExtns=*.txt,*.csv\0\0
\end{verbatim}

Suppose that a driver for SQL Server does not have a separate setup DLL and does not have any driver attribute keywords. The \textit{lpszDriver} argument for this driver might be as follows:

\begin{verbatim}
SQL\0Server\0Driver=SQLSRVR.DLL\0\0
\end{verbatim}

After \texttt{SQLInstallDriverEx} retrieves information about the driver from the \textit{lpszDriver} argument, it adds the driver description to the [ODBC Drivers] section of the Odbcinst.ini entry in the system information. It then creates a section titled with the driver's description and adds the full paths of the
driver DLL and the setup DLL. Finally, it returns the path of the target directory of the installation but
does not copy the driver files to it. The calling program must actually copy the driver files to the target
directory.

**SQLInstallDriverEx** increments the component usage count for the installed driver by 1. If a version
of the driver already exists but the component usage count for the driver does not exist, the new
component usage count value is set to 2.

The application setup program is responsible for physically copying the driver file and maintaining the
file usage count. If the driver file has not previously been installed, the application setup program
must copy the file in the *lpszPathIn* path and create the file usage count. If the file has previously
been installed, the setup program merely increments the file usage count and returns the path of the
prior installation in the *lpszPathOut* argument.

### Note

For more information about component usage counts and file usage counts, see [Usage Counting](#).

If an older version of the driver file was previously installed by the application, the driver should be
uninstalled and then reinstalled, so that the driver component usage count is valid. **SQLConfigDriver**
(with an *fRequest* of ODBC_REMOVE_DRIVER) should first be called, and then **SQLRemoveDriver**
should be called to decrement the component usage count. **SQLInstallDriverEx** should then be
called to reinstall the driver, incrementing the component usage count. The application setup program
must replace the old file with the new file. The file usage count will remain the same, and any other
application that used the older version file will now use the newer version.

### Note

If the driver was previously installed and **SQLInstallDriverEx** is called to install the driver in a
different directory, the function will return TRUE, but *lpszPathOut* will include the directory where
the driver was already installed. It will not include the directory entered in the *lpszDriver* argument.

The length of the path in *lpszPathOut* in **SQLInstallDriverEx** allows for a two-phase install process,
so an application can determine what *cbPathOutMax* should be by calling **SQLInstallDriverEx** with an
*fRequest* of ODBC_INSTALL_INQUIRY mode. This will return the total number of bytes available in the
*pcbPathOut* buffer. **SQLInstallDriverEx** can then be called with an *fRequest* of
ODBC_INSTALL_COMPLETE and the *cbPathOutMax* argument set to the value in the *pcbPathOut* buffer,
plus the null-termination character.

If you choose not to use the two-phase model for **SQLInstallDriverEx**, you must set *cbPathOutMax*,
which defines the size of the storage for the path of the target directory, to the value _MAX_PATH, as
defined in Stdlib.h, to prevent truncation.

When *fRequest* is ODBC_INSTALL_COMPLETE, **SQLInstallDriverEx** does not allow *lpszPathOut* to be
NULL (or *cbPathOutMax* to be 0). If *fRequest* is ODBC_INSTALL_COMPLETE, FALSE is returned when
the number of bytes available to return is greater than or equal to *cbPathOutMax*, with the result that
truncation occurs.

After **SQLInstallDriverEx** has been called and the application setup program has copied the driver
file (if necessary), the driver setup DLL must call **SQLConfigDriver** to set the configuration for the
driver.

### Related Functions
**SQLInstallDriverManager Function**

**Conformance**

Version Introduced: ODBC 1.0: Deprecated in Windows XP Service Pack 2, Windows Server 2003 Service Pack 1, and later operating systems

**Summary**

`SQLInstallDriverManager` returns the path of the target directory for the installation of the ODBC core components. The calling program must actually copy the Driver Manager’s files to the target directory.

**Syntax**

```c
BOOL SQLInstallDriverManager(
    LPSTR lpszPath,
    WORD cbPathMax,
    WORD * pcbPathOut);
```

**Arguments**

- **lpszPath**
  [Output] Path of the target directory of the installation.

- **cbPathMax**
  [Input] Length of `lpszPath`. This must be at least _MAX_PATH bytes.

- **pcbPathOut**
  [Output] Total number of bytes (excluding the null-termination byte) returned in `lpszPath`. If the number of bytes available to return is greater than or equal to `cbPathMax`, the path in `lpszPath` is truncated to `cbPathMax` minus the null-termination character. The `pcbPathOut` argument can be a null pointer.

**Returns**

The function returns TRUE if it is successful, FALSE if it fails.
Diagnostics

When `SQLInstallDriverManager` returns FALSE, an associated `*pfErrorCode` value can be obtained by calling `SQLInstallerError`. The following table lists the `*pfErrorCode` values that can be returned by `SQLInstallerError` and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_BUFF_LEN</td>
<td>Invalid buffer length</td>
<td>The <code>lpszPath</code> argument was not large enough to contain the output path.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The buffer contains the truncated path.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>cbPathMax</code> argument was less than <code>_MAX_PATH</code>.</td>
</tr>
<tr>
<td>ODBC_ERROR_USAGE_UPDATE_FAILED</td>
<td>Could not increment or decrement the</td>
<td>The installer failed to increment the ODBC core component usage count.</td>
</tr>
<tr>
<td></td>
<td>component usage count</td>
<td></td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

Comments

`SQLInstallDriverManager` is called to return the path for ODBC core components and increment the component usage count in the system information. If a version of the Driver Manager already exists but the component usage count for the driver does not exist, the new component usage count value is set to 2.

The application setup program is responsible for physically copying the core component files and maintaining the file usage counts. If a core component file has not previously been installed, the application setup program must copy the file, and create the file usage count. If the file has previously been installed, the setup program merely increments the file usage count.

If an older version of the Driver Manager was previously installed by the application setup program, the core components should be uninstalled and then reinstalled, so that the core component usage count is valid. `SQLRemoveDriverManager` should first be called to decrement the component usage count. `SQLInstallDriverManager` should then be called to increment the component usage count. The application setup program must replace the old core component files with the new files. The file usage counts will remain the same, and other applications that used the older version core component files will now use the newer version files.
In a fresh install of the ODBC core components, drivers, and translators, the application setup program should call the following functions in sequence: `SQLInstallDriverManager`, `SQLInstallDriverEx`, `SQLConfigDriver` (with an `fRequest` of ODBC_INSTALL_DRIVER), and then `SQLInstallTranslatorEx`. In an uninstall of the core components, drivers, and translators, the application setup program should call the following functions in sequence: `SQLRemoveTranslator`, `SQLRemoveDriver`, and then `SQLRemoveDriverManager`. These functions must be called in this sequence. In an upgrade of all components, all the uninstall functions should be called in sequence and then all the install functions should be called in sequence.

### Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding, modifying, or removing a driver</td>
<td>SQLConfigDriver</td>
</tr>
<tr>
<td>Installing a driver</td>
<td>SQLInstallDriverEx</td>
</tr>
<tr>
<td>Installing a translator</td>
<td>SQLInstallTranslatorEx</td>
</tr>
<tr>
<td>Removing a driver</td>
<td>SQLRemoveDriver</td>
</tr>
<tr>
<td>Removing the Driver Manager</td>
<td>SQLRemoveDriverManager</td>
</tr>
<tr>
<td>Removing a translator</td>
<td>SQLRemoveTranslator</td>
</tr>
</tbody>
</table>

### SQLInstallerError Function

#### Conformance

**Version Introduced:** ODBC 3.0

#### Summary

`SQLInstallerError` returns error or status information for the ODBC installer functions.

#### Syntax

```c
RETCODE SQLInstallerError(
    WORD     iError,
    DWORD *  pfErrorCode,
    LPSTR    lpszErrorMsg,
    WORD     cbErrorMsgMax,
    WORD *   pcbErrorMsg);
```
Arguments

iError

[Input] Error record number. Valid numbers are from 1 through 8.

pfErrorCode

[Output] Installer error code. (For more information, see "Comments.")

lpszErrorMsg

[Output] Pointer to storage for the error message text.

cbErrorMsgMax

[Input] Maximum length of the szErrorMsg buffer. This must be less than or equal to SQL_MAX_MESSAGE_LENGTH minus the null-termination character.

cbErrorMsgMax

[Input] Maximum length of the szErrorMsg buffer. This must be less than or equal to SQL_MAX_MESSAGE_LENGTH minus the null-termination character.

pcbErrorMsg

[Output] Pointer to the total number of bytes (excluding the null-termination character) available to return in lpszErrorMsg. If the number of bytes available to return is greater than or equal to cbErrorMsgMax, the error message text in lpszErrorMsg is truncated to cbErrorMsgMax minus the null-termination character bytes. The pcbErrorMsg argument can be a null pointer.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, or SQL_ERROR.

Diagnostics

SQLInstallerError does not post error values for itself. SQLInstallerError returns SQL_NO_DATA when it is unable to retrieve any error information (in which case pfErrorCode is undefined). If SQLInstallerError cannot access error values for any reason that would normally return SQL_ERROR, SQLInstallerError returns SQL_ERROR but does not post any error values. If you do not know the length of the warning string (lpszErrorMsg), you can set lpszErrorMsg to NULL and call SQLInstallerError. SQLInstallerError will then return the length of the warning string in cbErrorMsgMax. If the buffer for the error message is too short, SQLInstallerError returns SQL_SUCCESS_WITH_INFO and returns the correct pfErrorCode value for SQLInstallerError.

To determine whether a truncation occurred in the error message, an application can compare the value in the cbErrorMsgMax argument to the actual length of the message text written to the pcbErrorMsg argument. If truncation does occur, the correct buffer length should be allocated for lpszErrorMsg and SQLInstallerError should be called again with the corresponding iError record.

Comments
An application calls **SQLInstallerError** when a previous call to the ODBC installer function returns FALSE. ODBC installer and driver or translator setup functions post zero or more errors only when the function fails (returns FALSE); therefore, an application calls **SQLInstallerError** only after an ODBC installer function fails.

The ODBC installer error queue is flushed each time a new installer function is called. Therefore, an application cannot expect to retrieve errors for functions other than from the last installer function call.

To retrieve multiple errors for a function call, an application calls **SQLInstallerError** multiple times.

When there is no additional information, **SQLInstallerError** returns SQL_NO_DATA, the *pfErrorCode* argument is undefined, the *pcbErrorMsg* argument equals 0, and the *lpszErrorMsg* argument contains a single null-termination character (unless the *cbErrorMsgMax* argument is equal to 0).

---

**SQLInstallTranslator Function**

**Conformance**

Version Introduced: ODBC 2.5, Deprecated

**Summary**

In ODBC 3.0, **SQLInstallTranslator** has been replaced by **SQLInstallTranslatorEx**. Calls to **SQLInstallTranslator** will be mapped to **SQLInstallTranslatorEx**. For more information, see **SQLInstallTranslatorEx**.

**SQLInstallTranslator** will return FALSE if an application calls it in the ODBC 3.x Driver Manager with the *lpszInfFile* argument set to a value other than NULL. The Odbc.inf file used in ODBC 2.x is no longer supported in ODBC 3.x, even for backward compatibility.

---

**SQLInstallTranslatorEx Function**

**Conformance**

Version Introduced: ODBC 3.0

**Summary**

**SQLInstallTranslatorEx** adds information about a translator to the Odbcinst.ini section of the system information (HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Translators registry key).

The functionality of **SQLInstallTranslatorEx** can also be accessed with **ODBCCONF.EXE**.

**Syntax**

---
BOOL SQLInstallTranslatorEx(
    LPCSTR lpszTranslator,
    LPCSTR lpszPathIn,
    LPSTR lpszPathOut,
    WORD cbPathOutMax,
    WORD * pcbPathOut,
    WORD fRequest,
    LPDWORD lpdwUsageCount);

Arguments

lpszTranslator

[Input] This must contain a doubly null-terminated list of keyword-value pairs describing the translator. For more information about keyword-value pair syntax, see Translator Specification Subkeys.

The Translator and Setup keywords must be included in the lpszTranslator string. The translation DLL is listed with the Translator keyword, and the translator setup DLL is listed with the Setup keyword. Each pair is terminated with a NULL byte, and the entire list is terminated with a NULL byte. (That is, two NULL bytes mark the end of the list.) The format of lpszTranslator is as follows:

\0Translator=translator-DLL-filename\0[Setup=setup-DLL-filename\0]\0

lpszPathIn

[Input] Full path of where the translator is to be installed or a null pointer. If lpszPath is a null pointer, the translators will be installed in the System directory.

lpszPathOut

[Output] The path of the target directory where the translator should be installed. If the translator has never been installed, lpszPathOut is the same as lpszPathIn. If there exists a prior installation of the translator, lpszPathOut is the path of the prior installation.

cbPathOutMax

[Input] Length of lpszPathOut.

pcbPathOut

[Output] Total number of bytes available to return in lpszPathOut. If the number of bytes available to return is greater than or equal to cbPathOutMax, the output path in lpszPathOut is truncated to pcbPathOutMax minus the null-termination character. The pcbPathOut argument can be a null pointer.

fRequest

[Input] Type of request. fRequest must contain one of the following values:

    ODBC_INSTALL_INQUIRY: Inquire about where a translator can be installed.
    ODBC_INSTALL_COMPLETE: Complete the installation request.

lpdwUsageCount

[Output] The usage count of the translator after this function has been called.

Applications should not set the usage count. ODBC will maintain this count.
Returns

The function returns TRUE if it is successful, FALSE if it fails.

Diagnostics

When SQLInstallTranslatorEx returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_BUFF_LEN</td>
<td>Invalid buffer length</td>
<td>The lpszPathOut argument was not large enough to contain the output path. The buffer contains the truncated path. The cbPathOutMax argument was 0, and the fRequest argument was ODBC_INSTALL_COMPLETE.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_REQUEST_TYPE</td>
<td>Invalid type of request</td>
<td>The fRequest argument was not one of the following: ODBC_INSTALL_INQUIRY ODBC_INSTALL_COMPLETE</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_KEYWORD_VALUE</td>
<td>Invalid keyword-value pairs</td>
<td>The lpszTranslator argument contained a syntax error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_PATH</td>
<td>Invalid install path</td>
<td>The lpszPathIn argument contained an invalid path.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_PARAM_SEQUENCE</td>
<td>Invalid parameter sequence</td>
<td>The lpszTranslator argument did not contain a list of keyword-value pairs.</td>
</tr>
<tr>
<td>ODBC_ERROR_USAGE_UPDATE_FAILED</td>
<td>Could not increment or decrement the registry's component usage count</td>
<td>The installer failed to increment the translator's usage count.</td>
</tr>
</tbody>
</table>
Comments

**SQLInstallTranslatorEx** provides a mechanism to install just the translator. This function does not actually copy any files. The calling program is responsible for copying the translator files.

**SQLInstallTranslatorEx** increments the component usage count for the installed translator by 1. If a version of the translator already exists but the component usage count for the translator does not exist, the new component usage count value is set to 2.

The application setup program is responsible for physically copying the translator file and maintaining the file usage count. If the translator file has not previously been installed, the application setup program must copy the file or files and create the file or files usage count. If the file has previously been installed, the setup program simply increments the file usage count.

If an older version of the translator was previously installed by the application, the translator should be uninstalled and then reinstalled, so that the translator component usage count is valid. **SQLRemoveTranslator** should be called to decrement the component usage count, and then **SQLInstallTranslatorEx** should be called to increment the component usage count. The application setup program must replace the old file or files with the new file. The file usage count will remain the same, and other applications that used the older version file will now use the newer version.

The length of the path in *lpszPathOut* in **SQLInstallTranslatorEx** allows for a two-phase install process, so an application can determine what *cbPathOutMax* should be by calling **SQLInstallTranslatorEx** with an *fRequest* of ODBC_INSTALL_INQUIRY mode. This will return the total number of bytes available in the *pcbPathOut* buffer. **SQLInstallTranslatorEx** can then be called with an *fRequest* of ODBC_INSTALL_COMPLETE and the *cbPathOutMax* argument set to the value in the *pcbPathOut* buffer, plus the null-termination character.

If you choose not to use the two-phase model for **SQLInstallTranslatorEx**, you must set *cbPathOutMax*, which defines the size of the storage for the path of the target directory, to the value _MAX_PATH, as defined in Stdlib.h, to prevent truncation.

When *fRequest* is ODBC_INSTALL_COMPLETE, **SQLInstallTranslatorEx** does not allow *lpszPathOut* to be NULL (or *cbPathOutMax* to be 0). If *fRequest* is ODBC_INSTALL_COMPLETE, FALSE is returned when the number of bytes available to return is greater than or equal to *cbPathOutMax*, with the result that truncation occurs.

### Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returning a default translation option</td>
<td>ConfigTranslator</td>
</tr>
<tr>
<td>Selecting translators</td>
<td>SQLGetTranslator</td>
</tr>
<tr>
<td>Removing translators</td>
<td>SQLRemoveTranslator</td>
</tr>
</tbody>
</table>

© 2016 Microsoft
SQLManageDataSources

Conformance
Version Introduced: ODBC 2.0

Summary
SQLManageDataSources displays a dialog box with which users can set up, add, and delete data sources in the system information.

Syntax

```c
BOOL SQLManageDataSources(
    HWND hwnd);
```

Arguments

hwnd
[Input] Parent window handle.

Returns

SQLManageDataSources returns FALSE if hwnd is not a valid window handle. Otherwise, it returns TRUE.

Diagnostics

When SQLManageDataSources returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_REQUEST_FAILED</td>
<td>Request</td>
<td>The call to ConfigDSN failed.</td>
</tr>
<tr>
<td></td>
<td>failed</td>
<td></td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID__HWND</td>
<td>Invalid</td>
<td>The hwnd argument was invalid or NULL.</td>
</tr>
</tbody>
</table>
Managing Data Sources

**SQLManageDataSources** initially displays the **ODBC Data Source Administrator** dialog box, as shown in the following illustration.

The dialog box displays the data sources listed in the system information under three tabs: **User DSN**, **System DSN**, and **File DSN**. If the user double-clicks a data source or selects a data source and clicks **Configure**, **SQLManageDataSources** calls **ConfigDSN** in the setup DLL with the ODBC_CONFIG_DSN option.

If the user clicks **Add**, **SQLManageDataSources** displays the **Create New Data Source** dialog box, shown in the following illustration.
The dialog box displays a list of installed drivers. If the user double-clicks a driver or selects a driver and clicks **OK**, `SQLManageDataSources` calls `ConfigDSN` in the setup DLL and passes it the ODBC_ADD_DSN option.

If the user selects a data source and clicks **Remove**, `SQLManageDataSources` asks whether the user wants to delete the data source. If the user clicks **Yes**, `SQLManageDataSources` calls `ConfigDSN` in the setup DLL with the ODBC_REMOVE_DSN option.

The **Create New Data Source** dialog box is used to add or delete a user data source, a system data source, or a file data source.

**User DSNs**

DSNs created for individual users will be called User DSNs, to distinguish them from System DSNs. User DSNs are registered as follows in the system information:

```
HKEY_CURRENT_USERS
    SOFTWARE
        ODBC
            Odbc.ini
```

**System DSNs**

The **Create New Data Source** dialog box allows you to add a system data source to your local computer or delete one, or to set the configuration for a system data source.

A data source set up with a system data source name (DSN) can be used by more than one user on the same machine. It can also be used by a systemwide service, which can then gain access to the data source even if no user is logged on to the machine.
A System DSN is registered in the HKEY_LOCAL_MACHINE entry in the system information rather than in the HKEY_CURRENT_USER entry. It is not tied to one user who logs on with his or her particular user name and password but can be used by any user of that machine or by an automatic systemwide service. The System DSN is, however, tied to one machine. It does not support the capability of using remote DSNs between machines. System DSNs are registered as follows in the system information:

HKEY_LOCAL_MACHINE SOFTWARE ODBC Odbc.ini

File DSNs

A file data source does not have a data source name, as does a machine data source, and is not registered to any one user or machine. The connection information for that data source is contained in a .dsn file that can be copied to any machine. A file data source can be shareable, in which case the .dsn file resides on a network and can be used simultaneously by multiple users on the network as long as the user has the appropriate driver installed. A file data source can also be unshareable, in which case it can be used only on a single machine.

For more information on file data sources, see Connecting Using File Data Sources, or see SQLDriverConnect.

Managing Drivers

If the user clicks the Drivers tab in the ODBC Data Source Administrator dialog box, SQLManageDataSources displays a list of ODBC drivers installed on the system, as well as information about the drivers. The date displayed is the creation date of the driver, as shown in the following illustration.

![ODBC Data Source Administrator](image)

Tracing Options

![Tracing Options](image)
If the user clicks the Tracing tab in the ODBC Data Source Administrator dialog box, SQLManageDataSources displays tracing options, as shown in the following illustration.

If the user clicks Start Tracing Now and then clicks OK, SQLManageDataSources enables tracing manually for all applications currently running on the machine.

If the user specifies the name of a trace file in the Log file Path text box and then clicks OK, SQLManageDataSources sets the TraceFile keyword in the [ODBC] section of the system information to the specified name.

Important

Support for Visual Studio Analyzer was removed beginning in Windows 8 (Visual Studio Analyzer was only included in older versions of Visual Studio). For an alternative troubleshooting mechanism, use BID tracing.

If the user clicks Start Visual Studio Analyzer and then clicks OK, Visual Studio Analyzer is enabled. It remains enabled until Stop Visual Studio Analyzer is clicked.

For more information on tracing, see Tracing. For more information about the Trace and TraceFile keywords, see ODBC Subkey.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating data sources</td>
<td>SQLCreateDataSource</td>
</tr>
</tbody>
</table>
SQLPostInstallerError Function

Conformance
Version Introduced: ODBC 3.0

Summary
SQLPostInstallerError provides a mechanism for a driver or translator setup library to report errors for the ConfigDriver, ConfigDSN, and ConfigTranslator functions to the installer error queue. Applications do not use this API; they use SQLInstallerError to retrieve the error.

Syntax

```
RETCODE SQLPostInstallerError(
    DWORD fErrorCode,
    LPSTR szErrorMsg);
```

Arguments

`fErrorCode`
[Input] Installer error code.

`szErrorMsg`
[Input] Error message text.

Returns

SQL_SUCCESS or SQL_ERROR.

Diagnostics

SQLPostInstallerError does not post error values for itself. If the error was successfully posted to the installer error queue (retrievable using SQLInstallerError), SQL_SUCCESS is returned. SQL_ERROR will be returned if the value in the `dwErrorCode` argument is not one of the specified installer error codes.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
</table>

For information about
### SQLReadFileDSN Function

**Conformance**

Version Introduced: ODBC 3.0

**Summary**

`SQLReadFileDSN` reads information from a File DSN.

**Syntax**

```c
BOOL SQLReadFileDSN(
    LPCSTR lpszFileName,
    LPCSTR lpszAppName,
    LPCSTR lpszKeyName,
    LPSTR lpszString,
    WORD cbString,
    WORD * pcbString);
```

**Arguments**

- **lpszFileName**
  - [Input] Pointer to the data buffer containing the name of the .dsn file. A .dsn extension is appended to all file names that do not already have a .dsn extension. The value in `lpszFileName` must be a null-terminated string.

- **lpszAppName**
  - [Input] Pointer to the data buffer containing the name of the application. This is "ODBC" for the ODBC section. The value in `lpszAppName` must be a null-terminated string.

- **lpszKeyName**
  - [Input] Pointer to the data buffer containing the name of the key to be read. See "Comments" for reserved keywords. The value in `lpszKeyName` must be a null-terminated string.

- **lpszString**
  - [Output] Pointer to the data buffer containing the string associated with the key to be read.
If *lpszFileName is a valid .dsn file name but the lpszAppName argument is a null pointer and the lpszKeyName argument is a null pointer, then *lpszString contains a list of valid applications. If *lpszFileName is a valid .dsn file name and *lpszAppName is a valid application name, but the lpszKeyName argument is a null pointer, then *lpszString contains a list of valid reserved keywords in the appropriate section of the DSN file, delimited by semicolons. If *lpszFileName is a valid .dsn file name but *lpszAppName is a null pointer and the lpszKeyName argument is a null pointer, then *lpszString contains a list of the sections in the DSN file, delimited by semicolons.

**cbString**

[Input] Length of the *lpszString buffer.

**pcbString**

[Output] Total number of bytes available to return in *lpszString. If the number of bytes available to return is greater than or equal to cbString, the output string in *lpszString is truncated to cbString minus the null-termination character. The pcbString argument can be a null pointer.

**Returns**

The function returns TRUE if it is successful, FALSE if it fails.

**Diagnostics**

When SQLReadFileDSN returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_BUFF_LEN</td>
<td>Invalid buffer length</td>
<td>The lpszString argument was NULL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The cbString argument was less than or equal to 0.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_PATH</td>
<td>Invalid install path</td>
<td>The path of the file name specified in the lpszFileName argument was invalid.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_REQUEST_TYPE</td>
<td>Invalid type of request</td>
<td>The lpszAppName argument was NULL, while the lpszKeyName argument was valid.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>
ODBC_ERROR_OUTPUT_STRING_TRUNCATED | Output string truncated | The string returned in *lpszString was truncated because the value in cbString was less than or equal to the value in *pcbString.

ODBC_ERROR_REQUEST_FAILED | Request failed | The keyword did not exist in the file DSN.

**Comments**

ODBC reserves the section name [ODBC] in which to store the connection information. The reserved keywords for this section are the same as those reserved for a connect string in SQLDriverConnect. (For more information, see the SQLDriverConnect function description.)

Applications can use these reserved keywords to read the information in a File DSN. If an application wants to find out the DSN-less connection string associated with a File DSN, it can call SQLReadFileDSN for any of the reserved connection string keywords in the [ODBC] section. The full connection string passed in a DSN-less connection is a combination of all of the keywords (reserved and driver-specific) in the [ODBC] section.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing information to a File DSN</td>
<td>SQLWriteFileDSN</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**SQLRemoveDefaultDataSource Function**

**Conformance**

Version Introduced: ODBC 1.0, Deprecated

**Summary**

In ODBC 3.0, the SQLRemoveDefaultDataSource function has been replaced by a call to SQLConfigDataSource with an fRequest argument of ODBC_REMOVE_DEFAULT_DSN. If an ODBC 2.x installation program calls this function, the ODBC installer will map it to the following SQLConfigDataSource call:

```
    SQLConfigDataSource (NULL, ODBC_REMOVE_DEFAULT_DSN, NULL, NULL)
```
SQLRemoveDriver Function

**Conformance**
Version Introduced: ODBC 3.0

**Summary**
SQLRemoveDriver changes or removes information about the driver from the Odbcinst.ini entry in the system information.

**Syntax**

```c
BOOL SQLRemoveDriver(
    LPCSTR lpszDriver,
    BOOL fRemoveDSN,
    LPDWORD lpdwUsageCount);
```

**Arguments**

- `lpszDriver`  
  [Input] The name of the driver as registered in the Odbcinst.ini key of the system information.

- `fRemoveDSN`  
  [Input] The valid values are:
  
  TRUE: Remove DSNs associated with the driver specified in `lpszDriver`. FALSE: Do not remove DSNs associated with the driver specified in `lpszDriver`.

- `lpdwUsageCount`  
  [Output] The usage count of the driver after this function has been called.

**Returns**

The function returns TRUE if it is successful, FALSE if it fails. If no entry exists in the system information when this function is called, the function returns FALSE.

**Diagnostics**

When SQLRemoveDriver returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.
<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_COMPONENT_NOT_FOUND</td>
<td>Component not found in registry</td>
<td>The installer could not remove the driver information because it either did not exist in the registry or could not be found in the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_NAME</td>
<td>Invalid driver or translator name</td>
<td>The lpszDriver argument was invalid.</td>
</tr>
<tr>
<td>ODBC_ERROR_USAGE_UPDATE_FAILED</td>
<td>Could not increment or decrement the component usage count</td>
<td>The installer failed to decrement the usage count of the driver.</td>
</tr>
<tr>
<td>ODBC_ERROR_REQUEST_FAILED</td>
<td>Request failed</td>
<td>The fRemoveDSN argument was TRUE; however, one or more DSNs could not be removed. The call to SQLConfigDriver with the ODBC_REMOVE_DRIVER request failed.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

**Comments**

SQLRemoveDriver complements the SQLInstallDriverEx function and updates the component usage count in the system information. This function should be called only from a setup application.

SQLRemoveDriver will decrement the component usage count value by 1. If the component usage count goes to 0, the following will occur:

1. The SQLConfigDriver function with the ODBC_REMOVE_DRIVER option will be called. If the fRemoveDSN option is set to TRUE, the ConfigDSN function calls SQLRemoveDSNFromIni to remove all the data sources associated with the driver specified in lpszDriver. If the fRemoveDSN option is set to FALSE, the data sources will not be deleted.

2. The driver entry in the system information will be removed. The driver entry is in the following system information location, under the driver name:

```
HKEY_LOCAL_MACHINE
SOFTWARE
```
**ODBC**

**Odbcinst.ini**

*SQLRemoveDriver* does not actually remove any files. The calling program is responsible for deleting files and maintaining the file usage count. Only after both the component usage count and the file usage count have reached zero is a file physically deleted. Some files in a component can be deleted, and others not deleted, depending on whether the files are used by other applications that have incremented the file usage count.

*SQLRemoveDriver* is also called as part of an upgrade process. If an application detects that it has to perform an upgrade and it has previously installed the driver, the driver should be removed and then reinstalled. *SQLRemoveDriver* should first be called to decrement the component usage count, and then *SQLInstallDriverEx* should be called to increment the component usage count. The application setup program must replace the old files with the new files. The file usage count will remain the same, and other applications that use the older version files will now use the newer version.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding, modifying, or removing a driver</td>
<td><code>ConfigDriver</code> (in the Setup DLL)</td>
</tr>
<tr>
<td>Adding, modifying, or removing a driver</td>
<td><code>SQLConfigDriver</code></td>
</tr>
<tr>
<td>Installing a driver</td>
<td><code>SQLInstallDriverEx</code></td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**SQLRemoveDriverManager Function**

**Conformance**


**Summary**

*SQLRemoveDriverManager* changes or removes information about the ODBC core components from the Odbcinst.ini entry in the system information.

**Syntax**

```c
```
BOOL SQLRemoveDriverManager(
    LPDWORD pdwUsageCount);

Arguments

pdwUsageCount

[Output] The usage count of the Driver Manager after this function has been called.

Returns

The function returns TRUE if it is successful, FALSE if it fails. If no entry exists in the system information when this function is called, the function returns FALSE.

Diagnostics

When SQLRemoveDriverManager returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_COMPONENT_NOT_FOUND</td>
<td>Component not found in registry</td>
<td>The installer could not remove the Driver Manager information because it either did not exist in the registry or could not be found in the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_USAGE_UPDATE_FAILED</td>
<td>Could not increment or decrement the component usage count</td>
<td>The installer failed to decrement the usage count of the Driver Manager.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

Comments
**SQLRemoveDriverManager** complements the **SQLInstallDriverManager** function, and updates the component usage count in the system information. This function should be called only from a setup application.

**SQLRemoveDriverManager** will decrement the core component usage count by 1. If the component usage count goes to 0, the entry system information will be removed. The core component entry is in the following location in the system information, under the title "ODBC Core":

```
HKEY_LOCAL_MACHINE
  SOFTWARE
  ODBC
    Odbcinst.ini
```

⚠️ **Caution**

An application should not physically remove Driver Manager files when the component usage count and the file usage count reach zero.

**SQLRemoveDriverManager** does not actually remove any files. The calling program is responsible for deleting files and maintaining the file usage counts. Driver Manager files should not, however, be removed when both the component usage count and the file usage count have reached zero, because these files may be used by other applications that have not incremented the file usage count.

**SQLRemoveDriverManager** is called as part of the Uninstall process. ODBC core components (which include the Driver Manager, Cursor Library, Installer, Language Library, Administrator, thunking files, and so on) are uninstalled as a whole. The following files are not removed when **SQLRemoveDriverManager** is called as part of the Uninstall process:

<table>
<thead>
<tr>
<th>DLL1</th>
<th>DLL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC32DLL</td>
<td>ODBCCP32.DLL</td>
</tr>
<tr>
<td>ODBCCR32.DLL</td>
<td>ODBC16GT.DLL</td>
</tr>
<tr>
<td>ODBC32GT.DLL</td>
<td>ODBC32GT.DLL</td>
</tr>
<tr>
<td>ODBCINT.DLL</td>
<td>DS16GT.DLL</td>
</tr>
<tr>
<td>ODBCTRAC.DLL</td>
<td>DS32GT.DLL</td>
</tr>
<tr>
<td>MSVCRT40.DLL</td>
<td>ODBCAD32.EXE</td>
</tr>
<tr>
<td>ODBCCP32.CPL</td>
<td></td>
</tr>
</tbody>
</table>

**SQLRemoveDriverManager** is also called as part of an upgrade process. If an application detects that it has to perform an upgrade and it has previously installed the driver, the driver should be removed and then reinstalled.

**SQLRemoveDriverManager** should first be called to decrement the component usage count. **SQLInstallDriverEx** should then be called to increment the component usage count. The application setup program must replace the old core component files with the new files. The file usage counts will remain the same, and other applications that use the older version core component files will now use
Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installing a Driver Manager</td>
<td>SQLInstallDriverManager</td>
</tr>
</tbody>
</table>

SQLRemoveDSNFromIni Function

Conformance
Version Introduced: ODBC 1.0

Summary
SQLRemoveDSNFromIni removes a data source from the system information.

Syntax

```c
BOOL SQLRemoveDSNFromIni(
    LPCSTR lpszDSN);
```

Arguments

lpszDSN

[Input] Name of the data source to remove.

Returns

The function returns TRUE if it removes the data source or the data source was not in the Odbc.ini file. It returns FALSE if it fails to remove the data source.

Diagnostics

When SQLRemoveDSNFromIni returns FALSE, an associated *pfErrorCode value can be obtained by
calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_DSN</td>
<td>Invalid DSN</td>
<td>The lpszDSN argument was invalid.</td>
</tr>
<tr>
<td>ODBC_ERROR_REQUEST_FAILED</td>
<td>Request failed</td>
<td>The installer could not remove the DSN info from the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

**Comments**

SQLRemoveDSNFromIni removes the data source name from the [ODBC Data Sources] section of the system information. It also removes the data source specification section from the system information.

This function should be called only from a driver setup library.

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding, modifying, or removing a data source</td>
<td>ConfigDSN</td>
</tr>
<tr>
<td>Adding, modifying, or removing a data source</td>
<td>SQLConfigDataSource</td>
</tr>
<tr>
<td>Removing the default data source</td>
<td>SQLRemoveDefaultDataSource</td>
</tr>
<tr>
<td>Adding a data source name to the system information</td>
<td>SQLWriteDSNToIni</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**SQLRemoveTranslator Function**

**Conformance**

Version Introduced: ODBC 3.0
Summary

SQLRemoveTranslator removes information about a translator from the Odbcinst.ini section of the system information and decrements the translator’s component usage count by 1.

Syntax

```c
BOOL SQLRemoveTranslator(
    LPCSTR lpszTranslator,
    LPDWORD lpdwUsageCount);
```

Arguments

- **lpszTranslator**
  - [Input] The name of the translator as registered in the Odbcinst.ini key of the system information.

- **lpdwUsageCount**
  - [Output] The usage count of the translator after this function has been called.

Returns

The function returns TRUE if it is successful, FALSE if it fails. If no entry exists in the system information when this function is called, the function returns FALSE.

Diagnostics

When SQLRemoveTranslator returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_COMPONENT_NOT_FOUND</td>
<td>Component not found in registry</td>
<td>The installer could not remove the translator information because it either did not exist in the registry or could not be found in the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_NAME</td>
<td>Invalid driver or translator</td>
<td>The lpszTranslator argument was invalid.</td>
</tr>
<tr>
<td>name</td>
<td>Could not increment or decrement the component usage count</td>
<td>The installer failed to decrement the usage count of the driver.</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>----------------------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

Comments

**SQLRemoveTranslator** complements the **SQLInstallTranslatorEx** function and updates the component usage count in the system information. This function should be called only from a setup application.

**SQLRemoveTranslator** will decrement the component usage count by 1. If the component usage count goes to 0, the translator entry in the system information will be removed. The translator entry is in the following location in the system information, under the translator name:

```
HKEY_LOCAL_MACHINE
   \SOFTWARE
   \ODBC
       odbcinst.ini
```

**SQLRemoveTranslator** does not actually remove any files. The calling program is responsible for deleting files, and maintaining the file usage count. Only after both the component usage count and the file usage count have reached zero is a file physically deleted. Some files in a component can be deleted, and others not deleted, depending on whether the files are used by other applications that have incremented the file usage count.

**SQLRemoveTranslator** is also called as part of an upgrade process. If an application detects that it has to perform an upgrade and it has previously installed the driver, the driver should be removed and then reinstalled. **SQLRemoveTranslator** should first be called to decrement the component usage count, and then **SQLInstallTranslatorEx** should be called to increment the component usage count. The application setup program must physically replace the old files with the new files. The file usage count will remain the same, and other applications that use the older version files will now use the newer version.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installing a translator</td>
<td>SQLInstallTranslatorEx</td>
</tr>
</tbody>
</table>
SQLSetConfigMode Function

Conformance
Version Introduced: ODBC 3.0

Summary
SQLSetConfigMode sets the configuration mode that indicates where the Odbc.ini entry listing DSN values is in the system information.

Syntax

```c
BOOL SQLSetConfigMode(
    UWORD wConfigMode);
```

Arguments

wConfigMode
[Input] The installer configuration mode (see "Comments"). The value in wConfigMode can be:

- ODBC_USER_DSN
- ODBC_SYSTEM_DSN
- ODBC_BOTH_DSN

Returns

The function returns TRUE if it is successful, FALSE if it fails.

Diagnostics

When SQLSetConfigMode returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
</table>

Invalid parameter sequence

The wConfigMode argument did not contain ODBC_USER_DSN, ODBC_SYSTEM_DSN, or ODBC_BOTH_DSN.

Comments

This function is used to set where the Odbc.ini entry listing DSN values is in the system information. If wConfigMode is ODBC_USER_DSN, the DSN is a User DSN and the function reads from the Odbc.ini entry in HKEY_CURRENT_USER. If it is ODBC_SYSTEM_DSN, the DSN is a System DSN and the function reads from the Odbc.ini entry in HKEY_LOCAL_MACHINE. If it is ODBC_BOTH_DSN, HKEY_CURRENT_USER is tried, and if it fails, then HKEY_LOCAL_MACHINE is used.

This function does not affect SQLCreateDataSource and SQLDriverConnect. The configuration mode has to be set when a driver reads from the registry by calling SQLGetPrivateProfileString or writes to the registry by calling SQLWritePrivateProfileString. Calls to SQLGetPrivateProfileString and SQLWritePrivateProfileString use the configuration mode to know which part of the registry to operate on.

Caution

SQLSetConfigMode should be called only when necessary; if the mode is improperly set, the ODBC Installer may fail to function properly.

SQLSetConfigMode makes a direct registry modification of the configuration mode. This is apart from the process of modifying the configuration mode by a call to SQLConfigDataSource. A call to SQLConfigDataSource sets the configuration mode to distinguish user and System DSNs when modifying a DSN. Prior to returning, SQLConfigDataSource resets the configuration mode to BOTHDSN.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating a data source</td>
<td>SQLCreateDataSource</td>
</tr>
<tr>
<td>Connecting to a data source using a connection string or dialog box</td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>Retrieving the configuration mode</td>
<td>SQLGetConfigMode</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

SQLValidDSN Function
Conformance
Version Introduced: ODBC 2.0

Summary
SQLValidDSN checks the length and validity of the data source name before the name is added to the system information.

Syntax

```c
BOOL SQLValidDSN(
    LPCSTR lpszDSN);
```

Arguments

lpszDSN

[Input] Data source name to be checked.

Returns

The function returns TRUE if the data source name is valid. It returns FALSE if the data source name is invalid or the function call failed.

Diagnostics

When SQLValidDSN returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. A *pfErrorCode is returned only if the function call failed, not if FALSE was returned because the data source name is invalid. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

Comments

SQLValidDSN is called by a driver's ConfigDSN to check the length of the data source name.
and the validity of the individual characters in the data source name. It checks whether the length of the name is greater than SQL_MAX_DSN_LENGTH, as defined in Sqlext.h. (The length of the data source name is also checked by SQLWriteDSNToIni.) SQLValidDSN checks whether any of the following invalid characters are included in the data source name:

```
[ ] { } ( ) , ; ? * = ! @ \n```

**Related Functions**

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding, modifying, or removing a data source</td>
<td>ConfigDSN (in the Setup DLL)</td>
</tr>
<tr>
<td>Adding, modifying, or removing a data source</td>
<td>SQLConfigDataSource</td>
</tr>
<tr>
<td>Writing a data source name to the system information</td>
<td>SQLWriteDSNToIni</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

### SQLWriteDSNToIni Function

**Conformance**

Version Introduced: ODBC 1.0

**Summary**

SQLWriteDSNToIni adds a data source to the system information.

**Syntax**

```c
BOOL SQLWriteDSNToIni(
    LPCSTR lpszDSN,
    LPCSTR lpszDriver);
```

**Arguments**

- `lpszDSN`  
  [Input] Name of the data source to add.

- `lpszDriver`  
  [Input] Driver description (usually the name of the associated DBMS) presented to users instead of the physical driver name.
Returns

The function returns TRUE if it is successful, FALSE if it fails.

Diagnostics

When SQLWriteDSNToIni returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_DSN</td>
<td>Invalid DSN</td>
<td>The lpszDSN argument contained a string that was invalid for a DSN.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_NAME</td>
<td>Invalid driver or translator name</td>
<td>The lpszDriver argument was invalid.</td>
</tr>
<tr>
<td>ODBC_ERROR_REQUEST_FAILED</td>
<td>Request failed</td>
<td>The installer failed to create a DSN in the registry.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

Comments

SQLWriteDSNToIni adds the data source to the [ODBC Data Sources] section of the system information. It then creates a specification section for the data source and adds a single keyword (Driver) with the name of the driver DLL as its value. If the data source specification section already exists, SQLWriteDSNToIni removes the old section before creating the new one.

The caller of this function must add any driver-specific keywords and values to the data source specification section of the system information.

If the name of the data source is Default, SQLWriteDSNToIni also creates the default driver specification section in the system information.

This function should be called only from a setup DLL.

Related Functions
For information about

| Adding, modifying, or removing a data source | ConfigDSN (in the Setup DLL) |
| Adding, modifying, or removing a data source | SQLConfigDataSource |
| Removing a data source name from the system information | SQLRemoveDSNFromIni |

© 2016 Microsoft

SQLWriteFileDSN Function

Conformance
Version Introduced: ODBC 3.0

Summary
SQLWriteFileDSN writes information to a File DSN.

Syntax

```c
BOOL SQLWriteFileDSN(
    LPCSTR lpszFileName,
    LPCSTR lpszAppName,
    LPCSTR lpszKeyName,
    LPCSTR lpszString);
```

Arguments

- **lpszFileName**
  - [Input] Pointer to the name of the File DSN. A DSN extension is appended to all file names that do not already have a DSN extension.

- **lpszAppName**
  - [Input] Pointer to the name of the application. This is "ODBC" for the ODBC section.

- **lpszKeyName**
  - [Input] Pointer to the name of the key to be read. See "Comments" for reserved keywords.

- **lpszString**
  - [Output] Pointed to the string associated with the key to be written. The maximum length of the string pointed to by this argument is 32,767 bytes.
Returns

The function returns TRUE if it is successful, FALSE if it fails.

Diagnostics

When SQLWriteFileDSN returns FALSE, an associated *pfErrorCode value can be obtained by calling SQLInstallerError. The following table lists the *pfErrorCode values that can be returned by SQLInstallerError and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_PATH</td>
<td>Invalid install path</td>
<td>The path of the file name specified in the lpszFileName argument was invalid.</td>
</tr>
<tr>
<td>ODBC_ERROR_INVALID_REQUEST_TYPE</td>
<td>Invalid type of request</td>
<td>The lpszAppName, lpszKeyName, or lpszString argument was NULL.</td>
</tr>
</tbody>
</table>

Comments

ODBC reserves the section name [ODBC] in which to store the connection information. The reserved keywords for this section are the same as those reserved for a connect string in SQLDriverConnect. (For more information, see the SQLDriverConnect function description.)

Applications can use these reserved keywords to write information directly to a File DSN. If an application wants to create or modify the DSN-less connection string associated with a File DSN, it can call SQLWriteFileDSN for any of the reserved connection string keywords in the [ODBC] section.

If the lpszString argument is a null pointer, the keyword pointed to by the lpszKeyName argument will be deleted from the .dsn file. If the lpszString and lpszKeyName arguments are both null pointers, the section pointed to by the lpszAppName argument will be deleted from the .dsn file.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading information from File DSNs</td>
<td>SQLReadFileDSN</td>
</tr>
</tbody>
</table>
SQLWritePrivateProfileString Function

Conformance
Version Introduced: ODBC 2.0

Summary
SQLWritePrivateProfileString writes a value name and data to the Odbc.ini subkey of the system information.

Syntax

```
BOOL SQLWritePrivateProfileString(  
    LPCSTR lpszSection,  
    LPCSTR lpszEntry,    
    LPCSTR lpszString,   
    LPCSTR lpszFilename);
```

Arguments

- **lpszSection**
  [Input] Points to a null-terminated string containing the name of the section to which the string will be copied. If the section does not exist, it is created. The name of the section is case-independent; the string can be any combination of uppercase and lowercase letters.

- **lpszEntry**
  [Input] Points to a null-terminated string containing the name of the key to be associated with a string. If the key does not exist in the specified section, it is created. If this argument is NULL, the entire section, including all entries within the section, is deleted.

- **lpszString**
  [Input] Points to a null-terminated string to be written to the file. If this argument is NULL, the key pointed to by the lpszEntry argument is deleted.

- **lpszFilename**
  [Output] Points to a null-terminated string that names the initialization file.

Returns

The function returns TRUE if it is successful, FALSE if it fails.
Diagnostics

When `SQLWritePrivateProfileString` returns FALSE, an associated `*pfErrorCode` value can be obtained by calling `SQLInstallerError`. The following table lists the `*pfErrorCode` values that can be returned by `SQLInstallerError` and explains each one in the context of this function.

<table>
<thead>
<tr>
<th>*pfErrorCode</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC_ERROR_GENERAL_ERR</td>
<td>General installer error</td>
<td>An error occurred for which there was no specific installer error.</td>
</tr>
<tr>
<td>ODBC_ERROR_REQUEST_FAILED</td>
<td>Request failed</td>
<td>The requested system information could not be written.</td>
</tr>
<tr>
<td>ODBC_ERROR_OUT_OF_MEM</td>
<td>Out of memory</td>
<td>The installer could not perform the function because of a lack of memory.</td>
</tr>
</tbody>
</table>

Comments

`SQLWritePrivateProfileString` is provided as a simple way to port drivers and driver setup DLLs from Microsoft® Windows® to Microsoft Windows NT®/Windows 2000. Calls to `WritePrivateProfileString` that write a profile string to the Odbc.ini file should be replaced with calls to `SQLWritePrivateProfileString`. `SQLWritePrivateProfileString` calls functions in the Win32® API to add the specified value name and data to the Odbc.ini subkey of the system information.

A configuration mode indicates where the Odbc.ini entry listing DSN values is in the system information. If the DSN is a User DSN (the state variable is USERDSN_ONLY), the function writes to the Odbc.ini entry in HKEY_CURRENT_USER. If the DSN is a System DSN (SYSTEMDSN_ONLY), the function writes to the Odbc.ini entry in HKEY_LOCAL_MACHINE. If the state variable is BOTHDSN, HKEY_CURRENT_USER is tried, and if it fails, HKEY_LOCAL_MACHINE is used.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting a value from the system information</td>
<td><code>SQLGetPrivateProfileString</code></td>
</tr>
</tbody>
</table>

Translation DLL API Reference

This section describes the syntax of the translation DLL API, which consists of two functions:
**SQLDriverToDataSource** and **SQLDataSourceToDriver**. These functions must be included in the DLL that performs translation for the driver.

This section contains the following topics.

- **SQLDataSourceToDriver Function**
- **SQLDriverToDataSource Function**

© 2016 Microsoft

## SQLDataSourceToDriver Function

**SQLDataSourceToDriver** supportstranslations for ODBC drivers. This function is not called by ODBC-enabled applications; applications request translation through **SQLSetConnectAttr**. The driver associated with the **ConnectionHandle** specified in **SQLSetConnectAttr** calls the specified DLL to perform translations of all data flowing from the data source to the driver. A default translation DLL can be specified in the ODBC initialization file.

### Syntax

```c
BOOL SQLDataSourceToDriver(
    UDWORD fOption,
    SWORD fSqlType,
    PTR rgbValueIn,
    SDWORD cbValueIn,
    PTR rgbValueOut,
    SDWORD cbValueOutMax,
    SDWORD * pcbValueOut,
    UCHAR * szErrorMsg,
    SWORD cbErrorMsgMax,
    SWORD * pcbErrorMsg);
```

### Arguments

- **fOption**
  
  [Input] Option value.

- **fSqlType**
  
  [Input] The SQL data type. This argument tells the driver how to convert **rgbValueIn** into a form acceptable by the application. For a list of valid SQL data types, see the SQL Data Types section in Appendix D: Data Types.

- **rgbValueIn**
  
  [Input] Value to translate.
**cbValueIn**

[Input] Length of `rgbValueIn`. 

**rgbValueOut**


- **Note**
  The translation DLL does not null-terminate this value.

**cbValueOutMax**

[Input] Length of `rgbValueOut`.

**pcbValueOut**

[Output] The total number of bytes (excluding the null-termination byte) available to return in `rgbValueOut`.

- For character or binary data, if this is greater than or equal to `cbValueOutMax`, the data in `rgbValueOut` is truncated to `cbValueOutMax` bytes.
- For all other data types, the value of `cbValueOutMax` is ignored and the translation DLL assumes that the size of `rgbValueOut` is the size of the default C data type of the SQL data type specified with `fSqlType`.

- The `pcbValueOut` argument can be a null pointer.

**szErrorMsg**

[Output] Pointer to storage for an error message. This is an empty string unless the translation failed.

**cbErrorMsgMax**

[Input] Length of `szErrorMsg`.

**pcbErrorMsg**

[Output] Pointer to the total number of bytes (excluding the null-termination byte) available to return in `szErrorMsg`. If this is greater than or equal to `cbErrorMsg`, the data in `szErrorMsg` is truncated to `cbErrorMsgMax` minus the null-termination character. The `pcbErrorMsg` argument can be a null pointer.

**Returns**

TRUE if the translation was successful, FALSE if the translation failed.

**Comments**

The driver calls `SQLDataSourceToDriver` to translate all data (result set data, table names, row counts, error messages, and so on) passing from the data source to the driver. The translation DLL might not translate some data, depending on the data's type and the purpose of the translation DLL; for example, a DLL that translates character data from one code page to another ignores all numeric and binary data.

- The value of `fOption` is set to the value of `vParam` specified by calling `SQLSetConnectAttr` with the
SQL_ATTR_TRANSLATE_OPTION attribute. It is a 32-bit value that has a specific meaning for a given translation DLL. For example, it could specify a certain character set translation.

If the same buffer is specified for rgbValueIn and rgbValueOut, the translation of data in the buffer will be performed in place.

Although cbValueIn, cbValueOutMax, and pcbValueOut are of the type SDWORD, SQLDataSourceToDriver does not necessarily support huge pointers.

If SQLDataSourceToDriver returns FALSE, data truncation might have occurred during translation. If pcbValueOut (the number of bytes available to return in the output buffer) is greater than cbValueOutMax (the length of the output buffer), then truncation occurred. The driver must determine whether the truncation was acceptable. If truncation did not occur, the SQLDataSourceToDriver returned FALSE due to another error. In either case, a specific error message is returned in szErrorMsg.

For more information about translating data, see Translation DLLs.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translating data being sent to the data source</td>
<td>SQLDriverToDataSource</td>
</tr>
<tr>
<td>Returning the setting of a connection attribute</td>
<td>SQLGetConnectAttr</td>
</tr>
<tr>
<td>Setting a connection attribute</td>
<td>SQLSetConnectAttr</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**SQLDriverToDataSource Function**

SQLDriverToDataSource supports translations for ODBC drivers. This function is not called by ODBC-enabled applications; applications request translation through SQLSetConnectAttr. The driver associated with the ConnectionHandle specified in SQLSetConnectAttr calls the specified DLL to perform translations of all data flowing from the driver to the data source. A default translation DLL can be specified in the ODBC initialization file.

**Syntax**

```c
BOOL SQLDriverToDataSource(
    UDWORD fOption,
    SWORD fSqlType,
    PTR rgbValueIn,
    SDWORD cbValueIn,
    ...);
```
**Arguments**

*fOption*

[Input] Option value.

*fSqlType*

[Input] The ODBC SQL data type. This argument tells the driver how to convert *rgbValueIn* into a form acceptable by the data source. For a list of valid SQL data types, see SQL Data Types.

*rgbValueIn*

[Input] Value to translate.

*cbValueIn*

[Input] Length of *rgbValueIn*.

*rgbValueOut*


<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>The translation DLL does not null-terminate this value.</td>
</tr>
</tbody>
</table>

*cbValueOutMax*

[Input] Length of *rgbValueOut*.

*pcbValueOut*

[Output] The total number of bytes (excluding the null-termination byte) available to return in *rgbValueOut*. For character or binary data, if this is greater than or equal to *cbValueOutMax*, the data in *rgbValueOut* is truncated to *cbValueOutMax* bytes.

For all other data types, the value of *cbValueOutMax* is ignored and the translation DLL assumes that the size of *rgbValueOut* is the size of the default C data type of the SQL data type specified with *fSqlType*.

The *pcbValueOut* argument can be a null pointer.

*szErrorMsg*

[Output] Pointer to storage for an error message. This is an empty string unless the translation failed.

*cbErrorMsgMax*

[Input] Length of *szErrorMsg*.

*pcbErrorMsg*

[Output] Pointer to the total number of bytes (excluding the null-termination byte) available to return in *szErrorMsg*. If this is greater than or equal to *cbErrorMsg*, the data in *szErrorMsg* is
Returns

TRUE if the translation was successful, FALSE if the translation failed.

Comments

The driver calls SQLDriverToDataSource to translate all data (SQL statements, parameters, and so on) passing from the driver to the data source. The translation DLL might not translate some data, depending on the data's type and the purpose of the translation DLL. For example, a DLL that translates character data from one code page to another ignores all numeric and binary data.

The value of fOption is set to the value of vParam specified by calling SQLSetConnectAttr with the SQL_ATTR_TRANSLATE_OPTION attribute. It is a 32-bit value that has a specific meaning for a given translation DLL. For example, it could specify a certain character set translation.

If the same buffer is specified for rgbValueIn and rgbValueOut, the translation of data in the buffer will be performed in place.

Although cbValueIn, cbValueOutMax, and pcbValueOut are of the type SDWORD, SQLDriverToDataSource does not necessarily support huge pointers.

If SQLDriverToDataSource returns FALSE, data truncation might have occurred during translation. If pcbValueOut (the number of bytes available to return in the output buffer) is greater than cbValueOutMax (the length of the output buffer), then truncation occurred. The driver must determine whether or not the truncation was acceptable. If truncation did not occur, the SQLDriverToDataSource returned FALSE due to another error. In either case, a specific error message is returned in szErrorMsg.

For more information about translating data, see Translation DLLs.

Related Functions

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translating data returned from the data source</td>
<td>SQLDataSourceToDriver</td>
</tr>
<tr>
<td>Returning the setting of a connection attribute</td>
<td>SQLGetConnectAttr</td>
</tr>
<tr>
<td>Setting a connection attribute</td>
<td>SQLSetConnectAttr</td>
</tr>
</tbody>
</table>
ODBC Service Provider Interface (SPI) Reference

Traditionally, ODBC defined an application programming interface (API). The functions in the API can be called by applications and they should be implemented inside both the Driver Manager and the driver.

With the addition of the driver-aware connection pooling feature, ODBC introduces the service provider interface (SPI). The functions in the SPI are used for communication between the Driver Manager and driver. SPI functions are implemented by the driver; the Driver Manager does not expose SPI functions to applications. Applications should not call these functions directly.

Include sqlspi.h for ODBC driver development.

This section contains the following topics

- SQLCleanupConnectionPoolID
- SQLGetPoolID
- SQLPoolConnect
- SQLRateConnection
- SQLSetConnectAttrForDbcInfo
- SQLSetConnectInfo
- SQLSetDriverConnectInfo

See Also

- Developing an ODBC Driver
- Developing Connection-Pool Awareness in an ODBC Driver
- Driver Manager Connection Pooling

© 2016 Microsoft

SQLCleanupConnectionPoolID Function

Conformance
Version Introduced: ODBC 3.81 Standards Compliance: ODBC

Summary
SQLCleanupConnectionPoolID informs a driver that a pool ID was timed out. A pool ID can timeout whenever all connections in a pool associated with that pool ID were timed out. See Pooling in the Microsoft Data Access Components for more information about connection timeout.
Syntax

```c
SQLRETURN SQLCleanupConnectionPoolID (SQLHENV EnvironmentHandle, SQLPOOLID PoolID);
```

Arguments

- **EnvironmentHandle**
  - [Input] The environment handle of the pool.

- **PoolID**
  - [Input] The pool associated to the pool ID that was timed out.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

The Driver Manager will not process diagnostic information returned from SQLCleanupConnectionPoolID.

An application cannot receive the error message returned by the driver.

Remarks

SQLCleanupConnectionPoolID can be called at any time, but the Driver Manager guarantees that no other thread is simultaneously calling SQLGetPoolID and no other thread is simultaneously calling SQLRateConnection and SQLPoolConnect with a connection info token assigned with that pool ID. Therefore, the driver must make sure this function is thread safe.

A driver can clean up the resources associated with the pool ID.

Applications should not call this function directly. An ODBC driver that supports driver-aware connection pooling must implement this function.

Include sqlspi.h for ODBC driver development.

See Also
Developing an ODBC Driver
Driver-Aware Connection Pooling
Developing Connection-Pool Awareness in an ODBC Driver

© 2016 Microsoft

## SQLGetPoolID Function

### Conformance

Version Introduced: ODBC 3.81 Standards Compliance: ODBC

### Summary

SQLGetPoolID retrieves the pool ID.

### Syntax

```c
SQLRETURN SQLGetPoolID (    SQLDBC_INFO_TOKEN  hDbcInfoToken,
    POOLID *            pPoolID );
```

### Arguments

**hDbcInfoToken**

[Input] Token handle that contains all connection information.

**pPoolID**

[Output] The pool ID, which is used to identify a set of connections that can be used interchangeably (possibly requiring an additional reset).

### Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

### Diagnostics

When SQLGetPoolID returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, the Driver Manager will use a `HandleType` of SQL_HANDLE_DBC_INFO_TOKEN and a `Handle` of `hDbcInfoToken`.
Remarks

`SQLGetPoolID` is used to obtain the pool ID given a set of connection information (from `SQLSetConnectAttrForDbcInfo`, `SQLSetDriverConnectInfo`, and `SQLSetConnectInfo`). This pool ID is used to identify a set of connections that can be used interchangeably (possibly requiring an additional reset). The pool ID will be used to identify the connection pool for that group of connections.

Whenever a driver returns `SQL_ERROR` or `SQL_INVALID_HANDLE`, the Driver Manager returns the error to the application (in `SQLConnect` or `SQLDriverConnect`).

Whenever a driver returns `SQL_SUCCESS_WITH_INFO`, the Driver Manager will obtain the diagnostic information from `hDbcInfoToken`, and return `SQL_SUCCESS_WITH_INFO` to the application in `SQLConnect` and `SQLDriverConnect`.

Applications should not call this function directly. An ODBC driver that supports driver-aware connection pooling must implement this function.

Include sqlspi.h for ODBC driver development.

See Also

- Developing an ODBC Driver
- Driver-Aware Connection Pooling
- Developing Connection-Pool Awareness in an ODBC Driver

© 2016 Microsoft

SQLPoolConnect Function

Conformance

Version Introduced: ODBC 3.8 Standards Compliance: ODBC

Summary

`SQLPoolConnect` is used to create a new connection if no connection in the pool can be reused.

Syntax

```c
SQLRETURN SQLPoolConnect(
    SQLHDBC hDbc,
    SQLHDBC_INFO_TOKEN hDbcInfoToken,
    WCHAR * wszOutConnectString,
    SQLSMALLINT cchConnectStringBuffer,
    SQLSMALLINT * cchConnectStringLen
);```
Arguments

*hDbc*

[Input] The connection handle.

*hDbcInfoToken*

[Input] The token handle for the new application connection request.

*wszOutConnectString*

[Output] Pointer to a buffer for the completed connection string. Upon successful connection to the target data source, this buffer contains the completed connection string. Applications should allocate at least 1,024 characters for this buffer.

If *wszOutConnectString* is NULL, *cchConnectStringLen* will still return the total number of characters (excluding the null-termination character for character data) available to return in the buffer pointed to by *wszOutConnectString*.

*cchConnectStringBuffer*

[Input] Length of the *wszOutConnectString* buffer, in characters.

*cchConnectStringLen*

[Output] Pointer to a buffer in which to return the total number of characters (excluding the null-termination character) available to return in *wszOutConnectString*. If the number of characters available to return is greater than or equal to *cchConnectStringBuffer*, the completed connection string in *wszOutConnectString* is truncated to *cchConnectStringBuffer* minus the length of a null-termination character.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or, SQL_INVALID_HANDLE.

Diagnostics

Similar to *SQLDriverConnect* for any input validation error, except that the Driver Manager will use a *HandleType* of SQL_HANDLE_DBC_INFO_TOKEN and a *Handle* of *hDbcInfoToken*.

Remarks

The Driver Manager guarantees that the parent HENV handle of *hDbc* and *hDbcInfoToken* are the same.

Unlike *SQLDriverConnect*, there is no *DriverCompletion* argument to prompt users to enter connection information. A prompting dialog is disallowed in the pooling scenario.

Applications should not call this function directly. An ODBC driver that supports driver-aware connection pooling must implement this function.

Whenever a driver returns SQL_ERROR or SQL_INVALID_HANDLE, the Driver Manager returns the
Whenever a driver returns SQL_SUCCESS_WITH_INFO, the Driver Manager will obtain the diagnostic information from hDbcInfoToken, and return SQL_SUCCESS_WITH_INFO to the application in SQLConnect and SQLDriverConnect.

When an application uses SQLConnect, wszOutConnectString will be a NULL buffer (the last three parameters will all be set to NULL, 0, NULL). Otherwise, the driver must return the output connection string, which will be returned to application’s SQLDriverConnect Function call.

Include sqlspi.h for ODBC driver development.

See Also

Developing an ODBC Driver
Driver-Aware Connection Pooling
Developing Connection-Pool Awareness in an ODBC Driver

© 2016 Microsoft

**SQLRateConnection Function**

**Conformance**

Version Introduced: ODBC 3.81 Standards Compliance: ODBC

**Summary**

**SQLRateConnection** determines if a driver can reuse an existing connection in the connection pool.

**Syntax**

```c
SQLRETURN SQLRateConnection(
    SQLHDBC_INFO_TOKEN hRequest,
    SQLHDBC hCandidateConnection,
    BOOL fRequiredTransactionEnlistment,
    TRANSID transId,
    DWORD * pRating);
```

**Arguments**

*hRequest*

[Input] A token handle representing the new application connection request.

*hCandidateConnection*
[Input] The existing connection in the connection pool. The connection must be in an opened state.

\textit{fRequiredTransactionEnlistment}

[Input] If TRUE, reusing the existing connection’s \textit{hCandidateConnection} for the new connection request (\textit{hRequest}) requires an additional enlistment.

\textit{transId}

[Input] If \textit{fRequiredTransactionEnlistment} is TRUE, \textit{transId} represents the DTC transaction that the request will enlist. If \textit{fRequiredTransactionEnlistment} is FALSE, \textit{transId} will be ignored.

\textit{pRating}

[Output] \textit{hCandidateConnection}’s reuse rating for the \textit{hRequest}. This rating will be in between 0 and 100 (inclusive).

**Returns**

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE.

**Diagnostics**

The Driver Manager will not process diagnostic information returned from this function.

**Remarks**

\textbf{SQLRateConnection} produces a score between 0 and 100 (inclusive) indicating how well an existing connection matches the request.

<table>
<thead>
<tr>
<th>Score</th>
<th>Meaning (when SQL_SUCCESS is returned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\textit{hCandidateConnection} must not be reused for the \textit{hRequest}.</td>
</tr>
<tr>
<td>Any values between 1 and 98 (inclusive)</td>
<td>The higher the score, the closer that \textit{hCandidateConnection} match with \textit{hRequest}.</td>
</tr>
<tr>
<td>99</td>
<td>There are only mismatches in insignificant attributes. The Driver Manager should stop the rating loop.</td>
</tr>
<tr>
<td>100</td>
<td>Perfect match. The Driver Manager should stop the rating loop.</td>
</tr>
<tr>
<td>Any other value greater than 100</td>
<td>\textit{hCandidateConnection} is marked as dead and it will not be reused even in an future connection request.</td>
</tr>
</tbody>
</table>

The Driver Manager will mark a connection as dead if the return code is anything other than SQL_SUCCESS (including SQL_SUCCESS_WITH_INFO) or the rating is greater than 100. That dead connection will not be reused (even in future connection requests) and will eventually be timed out.
after CPTimeout passes. The Driver Manager will continue to find another connection from the pool to rate.

If the Driver Manager reused a connection whose score is strictly smaller than 100 (including 99), the Driver Manager will call SQLSetConnectAttr(SQL_ATTR_DBC_INFO_TOKEN) to reset the connection back into the state requested by the application. The driver should not reset the connection in this function call.

If fRequiredTransactionEnlistment is TRUE, reusing hCandidateConnection needs an extra enlistment (transId != NULL) or unenlistment (transId == NULL). This indicates the cost of reusing a connection and whether the driver should enlist / unenlist the connection if it is going to reuse the connection. If fRequiredTransactionEnlistment is FALSE, driver should ignore the value of transId.

The Driver Manager guarantees that the parent HENV handle of hRequest and hCandidateConnection are the same. The Driver Manager guarantees that the pool ID associated with hRequest and hCandidateConnection are the same.

Applications should not call this function directly. An ODBC driver that supports driver-aware connection pooling must implement this function.

Include sqlspi.h for ODBC driver development.

See Also

Developing an ODBC Driver
Driver-Aware Connection Pooling
Developing Connection-Pool Awareness in an ODBC Driver

© 2016 Microsoft

SQLSetConnectAttrForDbcInfo Function

Conformance
Version Introduced: ODBC 3.81 Standards Compliance: ODBC

Summary
SQLSetConnectAttrForDbcInfo is the same as SQLSetConnectAttr, but it sets the attribute on the connection information token instead of on the connection handle.

Syntax

```
SQLRETURN SQLSetConnectAttrForDbcInfo(
    SQLHDBC_INFO_TOKEN hDbcInfoToken,
    SQLINTEGER Attribute,
    SQLPOINTER ValuePtr,
    SQLINTEGER StringLength );
```
Arguments

hDbcInfoToken
[Input] Token handle.

Attribute
[Input] Attribute to set. The list of valid attributes is driver specific and the same as for SQLSetConnectAttr.

ValuePtr
[Input] Pointer to the value to be associated with Attribute. Depending on the value of Attribute, ValuePtr will be a 32-bit unsigned integer value or will point to a null-terminated character string. Note that if the Attribute argument is a driver-specific value, the value in ValuePtr may be a signed integer.

StringLength
[Input] If Attribute is an ODBC-defined attribute and ValuePtr points to a character string or a binary buffer, this argument should be the length of *ValuePtr. For character string data, this argument should contain the number of bytes in the string.

If Attribute is an ODBC-defined attribute and ValuePtr is an integer, StringLength is ignored.

If Attribute is a driver-defined attribute, the application indicates the nature of the attribute to the Driver Manager by setting the StringLength argument. StringLength can have the following values:

- If ValuePtr is a pointer to a character string, then StringLength is the length of the string or SQL_NTSS.
- If ValuePtr is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in StringLength. This places a negative value in StringLength.
- If ValuePtr is a pointer to a value other than a character string or a binary string, then StringLength should have the value SQL_IS_POINTER.
- If ValuePtr contains a fixed-length value, then StringLength is either SQL_IS_INTEGER or SQL_IS_UINT, as appropriate.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

Same as SQLSetConnectAttr, except that the Driver Manager will use a HandleType of SQL_HANDLE_DBC_INFO_TOKEN and a Handle of hDbcInfoToken.
SQLSetConnectAttrForDbcInfo is the same as SQLSetConnectAttr, but it sets the attribute on the connection information token, instead of on the connection handle. For example, if SQLSetConnectAttr does not recognize an attribute, SQLSetConnectAttrForDbcInfo should also return SQL_ERROR for that attribute.

Whenever driver returns SQL_ERROR or SQL_INVALID_HANDLE, the driver should ignore this attribute to compute the pool ID. Also, the Driver Manager will obtain the diagnostic information from hDbcInfoToken, and return SQL_SUCCESS_WITH_INFO to the application in SQLConnect and SQLDriverConnect. Therefore, an application can retrieve details about why some attributes cannot be set.

Applications should not call this function directly. An ODBC driver that supports driver-aware connection pooling must implement this function.

Include sqlspi.h for ODBC driver development.

See Also

Developing an ODBC Driver
Driver-Aware Connection Pooling
Developing Connection-Pool Awareness in an ODBC Driver

© 2016 Microsoft

SQLSetConnectInfo Function

Conformance
Version Introduced: ODBC 3.81 Standards Compliance: ODBC

Summary
SQLSetConnectInfo is used to set the data source, user ID, and password into the connection info token for an application's SQLConnect call.

Syntax

```c
SQLRETURN SQLSetConnectInfo(
    SQLHDBC_INFO_TOKEN TokenHandle,
    WCHAR * ServerName,
    SQLSMALLINT NameLength1,
    WCHAR * UserName,
    SQLSMALLINT NameLength2,
    WCHAR * Authentication,
    SQLSMALLINT NameLength3 );
```
Arguments

**TokenHandle**
- [Input] Token handle.

**ServerName**
- [Input] Data source name. The data might be located on the same computer as the program, or on another computer somewhere on a network. For information about how an application chooses a data source, see Choosing a Data Source or Driver.

**NameLength1**
- [Input] Length of *ServerName* in characters.

**UserName**
- [Input] User identifier.

**NameLength2**
- [Input] Length of *UserName* in characters.

**Authentication**
- [Input] Authentication string (typically the password).

**NameLength3**
- [Input] Length of *Authentication* in characters.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

Same as **SQLConnect** for input validation errors, except that the Driver Manager will use a **HandleType** of SQL_HANDLE_DBC_INFO_TOKEN and a **Handle** of hDbcInfoToken.

Remarks

Whenever a driver returns SQL_ERROR or SQL_INVALID_HANDLE, the Driver Manager returns the error to the application (in **SQLConnect** or **SQLDriverConnect**).

Whenever a driver returns SQL_SUCCESS_WITH_INFO, the Driver Manager will obtain the diagnostic information from hDbcInfoToken, and return SQL_SUCCESS_WITH_INFO to the application in **SQLConnect** and **SQLDriverConnect**.

Applications should not call this function directly. An ODBC driver that supports driver-aware connection pooling must implement this function.

Include sqlspi.h for ODBC driver development.
See Also

- Developing an ODBC Driver
- Driver-Aware Connection Pooling
- Developing Connection-Pool Awareness in an ODBC Driver

© 2016 Microsoft

## SQLSetDriverConnectInfo Function

### Conformance

**Version Introduced:** ODBC 3.81  
**Standards Compliance:** ODBC

### Summary

`SQLSetDriverConnectInfo` is used to set the connection string into the connection info token for an application’s `SQLDriverConnect` call.

### Syntax

```c
SQLRETURN SQLSetDriverConnectInfo(
    SQLHDBC_INFO_TOKEN hDbcInfoToken,
    WCHAR * InConnectionString,
    SQLSMALLINT StringLength1
);```

### Arguments

- **TokenHandle**  
  [Input] Token handle.

- **InConnectionString**  
  [Input] A full connection string (see the syntax in "Comments" in `SQLDriverConnect`), a partial connection string, or an empty string.

- **StringLength1**  
  [Input] Length of `InConnectionString`, in characters if the string is Unicode, or bytes if string is ANSI or DBCS.

### Returns

`SQL_SUCCESS`, `SQL_SUCCESS_WITH_INFO`, `SQL_ERROR`, or `SQL_INVALID_HANDLE`.
Diagnostics

Same as SQLDriverConnect related to any input validation error, except that the Driver Manager will use a HandleType of SQL_HANDLE_DBC_INFO_TOKEN and a Handle of hDbcInfoToken.

Remarks

Whenever a driver returns SQL_ERROR or SQL_INVALID_HANDLE, the Driver Manager returns the error to the application (in SQLConnect or SQLDriverConnect).

Whenever a driver returns SQL_SUCCESS_WITH_INFO, the Driver Manager will obtain the diagnostic information from hDbcInfoToken, and return SQL_SUCCESS_WITH_INFO to the application in SQLConnect and SQLDriverConnect.

Applications should not call this function directly. An ODBC driver that supports driver-aware connection pooling must implement this function.

Include sqlspi.h for ODBC driver development.

See Also

Developing an ODBC Driver
Driver-Aware Connection Pooling
Developing Connection-Pool Awareness in an ODBC Driver

© 2016 Microsoft
ODBC Appendixes

Technical details relating to the Microsoft ODBC interface are contained in the following appendixes:

- Appendix A: ODBC Error Codes
- Appendix B: ODBC State Transition Tables
- Appendix C: SQL Grammar
- Appendix D: Data Types
- Appendix E: Scalar Functions
- Appendix F: ODBC Cursor Library
- Appendix G: Driver Guidelines for Backward Compatibility

© 2016 Microsoft

Appendix A: ODBC Error Codes

This topic discusses SQLSTATE values for ODBC 3.x. For more information on ODBC 3.x SQLSTATE values, see SQLSTATE Mappings.

SQLGetDiagRec or SQLGetDiagField returns SQLSTATE values as defined by Open Group Data Management: Structured Query Language (SQL), Version 2 (March 1995). SQLSTATE values are strings that contain five characters. The following table lists SQLSTATE values that a driver can return for SQLGetDiagRec.

The character string value returned for an SQLSTATE consists of a two-character class value followed by a three-character subclass value. A class value of "01" indicates a warning and is accompanied by a return code of SQL_SUCCESS_WITH_INFO. Class values other than "01," except for the class "IM," indicate an error and are accompanied by a return value of SQL_ERROR. The class "IM" is specific to warnings and errors that derive from the implementation of ODBC itself. The subclass value "000" in any class indicates that there is no subclass for that SQLSTATE. The assignment of class and subclass values is defined by SQL-92.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Error</th>
<th>Can be returned from</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>General warning</td>
<td>All ODBC functions except:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetDiagRec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetDiagField</td>
</tr>
<tr>
<td>01001</td>
<td>Cursor operation conflict</td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLExecute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLParamData</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLSetPos</td>
</tr>
<tr>
<td>01002</td>
<td>Disconnect error</td>
<td>SQLDisconnect</td>
</tr>
<tr>
<td>01003</td>
<td>NULL value eliminated in set function</td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLExecute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLParamData</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right-truncated</td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLBulkOperations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLColAttribute</td>
</tr>
</tbody>
</table>

Note: Although successful execution of a function is normally indicated by a return value of SQL_SUCCESS, the SQLSTATE 00000 also indicates success.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>01006</td>
<td>Privilege not revoked</td>
<td>SQLExecDirect, SQLExecute, SQLParamData</td>
</tr>
<tr>
<td>01007</td>
<td>Privilege not granted</td>
<td>SQLExecDirect, SQLExecute, SQLParamData</td>
</tr>
<tr>
<td>01S00</td>
<td>Invalid connection string attribute</td>
<td>SQLBrowseConnect, SQLDriverConnect</td>
</tr>
<tr>
<td>01S01</td>
<td>Error in row</td>
<td>SQLBulkOperations, SQLExtendedFetch, SQLSetPos</td>
</tr>
<tr>
<td>01S02</td>
<td>Option value changed</td>
<td>SQLBrowseConnect, SQLConnect, SQLDriverConnect, SQLExecDirect, SQLExecute, SQLParamData, SQLPParameter, SQLPrepare, SQLSetConnectAttr, SQLSetDescField, SQLSetEnvAttr, SQLSetStmtAttr</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>SQL Functions</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>01S06</td>
<td>Attempt to fetch before the result set returned the first rowset</td>
<td>( \text{SQLExtendedFetch} )&lt;br&gt;( \text{SQLFetchScroll} )</td>
</tr>
<tr>
<td>01S07</td>
<td>Fractional truncation</td>
<td>( \text{SQLBulkOperations} )&lt;br&gt;( \text{SQLExecDirect} )&lt;br&gt;( \text{SQLExecute} )&lt;br&gt;( \text{SQLExtendedFetch} )&lt;br&gt;( \text{SQLFetch} )&lt;br&gt;( \text{SQLFetchScroll} )&lt;br&gt;( \text{SQLGetData} )&lt;br&gt;( \text{SQLParamData} )&lt;br&gt;( \text{SQLSetPos} )</td>
</tr>
<tr>
<td>01S08</td>
<td>Error saving File DSN</td>
<td>( \text{SQLDriverConnect} )</td>
</tr>
<tr>
<td>01S09</td>
<td>Invalid keyword</td>
<td>( \text{SQLDriverConnect} )</td>
</tr>
<tr>
<td>07001</td>
<td>Wrong number of parameters</td>
<td>( \text{SQLExecDirect} )&lt;br&gt;( \text{SQLExecute} )</td>
</tr>
<tr>
<td>07002</td>
<td>COUNT field incorrect</td>
<td>( \text{SQLExecDirect} )&lt;br&gt;( \text{SQLExecute} )&lt;br&gt;( \text{SQLParamData} )</td>
</tr>
<tr>
<td>07005</td>
<td>Prepared statement not a cursor-specification</td>
<td>( \text{SQLColAttribute} )&lt;br&gt;( \text{SQLDescribeCol} )</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
<td>( \text{SQLBindCol} )&lt;br&gt;( \text{SQLBindParameter} )&lt;br&gt;( \text{SQLBulkOperations} )&lt;br&gt;( \text{SQLExecDirect} )&lt;br&gt;( \text{SQLExecute} )&lt;br&gt;( \text{SQLExtendedFetch} )&lt;br&gt;( \text{SQLFetch} )&lt;br&gt;( \text{SQLFetchScroll} )&lt;br&gt;( \text{SQLGetData} )&lt;br&gt;( \text{SQLParamData} )&lt;br&gt;( \text{SQLPutData} )&lt;br&gt;( \text{SQLSetPos} )</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
<td>( \text{SQLBindCol} )&lt;br&gt;( \text{SQLBindParameter} )&lt;br&gt;( \text{SQLBulkOperations} )&lt;br&gt;( \text{SQLColAttribute} )&lt;br&gt;( \text{SQLDescribeCol} )&lt;br&gt;( \text{SQLDescribeParam} )&lt;br&gt;( \text{SQLFetch} )&lt;br&gt;( \text{SQLFetchScroll} )&lt;br&gt;( \text{SQLGetData} )&lt;br&gt;( \text{SQLGetDescField} )&lt;br&gt;( \text{SQLGetDescRec} )</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Related Functions</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>07001</td>
<td>Invalid use of default parameter</td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td>07001</td>
<td>Invalid use of default parameter</td>
<td>SQLExecute</td>
</tr>
<tr>
<td>07001</td>
<td>Invalid use of default parameter</td>
<td>SQLParamData</td>
</tr>
<tr>
<td>07001</td>
<td>Invalid use of default parameter</td>
<td>SQLPutData</td>
</tr>
<tr>
<td>08001</td>
<td>Client unable to establish connection</td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td>08001</td>
<td>Client unable to establish connection</td>
<td>SQConnect</td>
</tr>
<tr>
<td>08001</td>
<td>Client unable to establish connection</td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>08002</td>
<td>Connection name in use</td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td>08002</td>
<td>Connection name in use</td>
<td>SQConnect</td>
</tr>
<tr>
<td>08002</td>
<td>Connection name in use</td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>08002</td>
<td>Connection name in use</td>
<td>SQLSetConnectAttr</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
<td>SQLAllocHandle</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
<td>SQLConnect</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>08003</td>
<td>Connection not open</td>
<td>SQLSetConnectAttr</td>
</tr>
<tr>
<td>08004</td>
<td>Server rejected the connection</td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td>08004</td>
<td>Server rejected the connection</td>
<td>SQConnect</td>
</tr>
<tr>
<td>08004</td>
<td>Server rejected the connection</td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>08007</td>
<td>Connection failure during transaction</td>
<td>SQLEndTran</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLColumnPrivileges</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLColumns</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQConnect</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLCopyDesc</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLDescribeCol</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLDescribeParam</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLExecute</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLFetch</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLForeignKeys</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLGetConnectAttr</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLGetData</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLGetDescField</td>
</tr>
<tr>
<td>08501</td>
<td>Communication link failure</td>
<td>SQLGetDescRec</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Functions</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>21S01</td>
<td>Insert value list does not match column list</td>
<td>SQLExecDirect, SQLPrepare</td>
</tr>
<tr>
<td>21S02</td>
<td>Degree of derived table does not match column list</td>
<td>SQLBulkOperations, SQLExecDirect, SQLExecute,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLParamData, SQLPrepare</td>
</tr>
<tr>
<td>22001</td>
<td>String data, right-truncated</td>
<td>SQLBulkOperations, SQLExecDirect, SQLExecute,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetch, SQLFetchScroll, SQLParamData, SQLPutData, SQLPutData, SQLSetDescField, SQLSetPos</td>
</tr>
<tr>
<td>22002</td>
<td>Indicator variable required but not supplied</td>
<td>SQLExecDirect, SQLExecute, SQLExtendedFetch,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetch</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>SQL Functions</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>22003</td>
<td>Numeric value out of range</td>
<td>SQLBulkOperations, SQLExecDirect, SQLExtendedFetch, SQLFetch, SQLFetchScroll, SQLGetData, SQLParamData, SQLPutData, SQLSetPos</td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format</td>
<td>SQLBulkOperations, SQLExecDirect, SQLExtendedFetch, SQLFetch, SQLFetchScroll, SQLGetData, SQLParamData, SQLPutData, SQLSetPos</td>
</tr>
<tr>
<td>22008</td>
<td>Datetime field overflow</td>
<td>SQLBulkOperations, SQLExecDirect, SQLParamData, SQLPutData</td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
<td>SQLExecDirect, SQLExtendedFetch, SQLFetch, SQLFetchScroll, SQLGetData, SQLPutData</td>
</tr>
<tr>
<td>22015</td>
<td>Interval field overflow</td>
<td>SQLExecDirect, SQLExtendedFetch, SQLFetch, SQLFetchScroll, SQLGetData, SQLParamData, SQLPutData</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Functions</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast specification</td>
<td>SQLBulkOperations, SQLExecDirect, SQLExecute, SQLExtendedFetch, SQLFetch, SQLFetchScroll, SQLGetData, SQLParamData, SQLPutData, SQLSetPos</td>
</tr>
<tr>
<td>22019</td>
<td>Invalid escape character</td>
<td>SQLExecDirect, SQLExecute, SQLPrepare</td>
</tr>
<tr>
<td>22025</td>
<td>Invalid escape sequence</td>
<td>SQLExecDirect, SQLExecute, SQLPrepare</td>
</tr>
<tr>
<td>22026</td>
<td>String data, length mismatch</td>
<td>SQLParamData</td>
</tr>
<tr>
<td>23000</td>
<td>Integrity constraint violation</td>
<td>SQLBulkOperations, SQLExecDirect, SQLExecute, SQLParamData, SQLSetPos</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
<td>SQLBulkOperations, SQLCloseCursor, SQLColumnPrivileges, SQLColumns, SQLExecDirect, SQLExecute, SQLExtendedFetch, SQLFetch, SQLFetchScroll, SQLForeignKeys, SQLGetData, SQLGetStmtAttr, SQLGetTypeInfo, SQLNativeSql, SQLPrepare, SQLPrimaryKeys, SQLProcedureColumns, SQLProcedures, SQLSetConnectAttr, SQLSetCursorName</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Functions</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>25000</td>
<td>Invalid transaction state</td>
<td>SQLDisconnect</td>
</tr>
<tr>
<td>25501</td>
<td>Transaction state</td>
<td>SQLEndTran</td>
</tr>
<tr>
<td>25502</td>
<td>Transaction is still active</td>
<td>SQLEndTran</td>
</tr>
<tr>
<td>25503</td>
<td>Transaction is rolled back</td>
<td>SQLEndTran</td>
</tr>
<tr>
<td>28000</td>
<td>Invalid authorization specification</td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>34000</td>
<td>Invalid cursor name</td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLPrepare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLSetCursorName</td>
</tr>
<tr>
<td>3C000</td>
<td>Duplicate cursor name</td>
<td>SQLSetCursorName</td>
</tr>
<tr>
<td>3D000</td>
<td>Invalid catalog name</td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLPrepare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLSetConnectAttr</td>
</tr>
<tr>
<td>3F000</td>
<td>Invalid schema name</td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLPrepare</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
<td>SQLBulkOperations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLColumnPrivileges</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLColumns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLEndTran</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLExecute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQFetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLForeignKeys</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetTypeInfo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLMoreResults</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLParamData</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLPrimaryKeys</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLProcedureColumns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLProcedures</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLSetPos</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLSpecialColumns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLStatistics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLTablePrivileges</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLTables</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Functions</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>40002</td>
<td>Integrity constraint violation</td>
<td>SQLEndTran</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
<td>SQLEndTran, SQLColumnPrivileges, SQLColumns, SQLEXecDirect, SQLExecute, SQLFetch, SQLFetchScroll, SQLForeignKeys, SQLGetTypeInfo, SQLMoreResults, SQLPrimaryKeys, SQLProcedureColumns, SQLProcedures, SQLParamData, SQLSetPos, SQLSpecialColumns, SQLStatistics, SQLTablePrivileges, SQLTables</td>
</tr>
<tr>
<td>42000</td>
<td>Syntax error or access violation</td>
<td>SQLEndTran, SQLEexecDirect, SQLExecute, SQLParamData, SQLPrepare, SQLSetPos</td>
</tr>
<tr>
<td>42S01</td>
<td>Base table or view already exists</td>
<td>SQLEndTran, SQLPrepare</td>
</tr>
<tr>
<td>42S02</td>
<td>Base table or view not found</td>
<td>SQLEndTran, SQLPrepare</td>
</tr>
<tr>
<td>42S11</td>
<td>Index already exists</td>
<td>SQLEndTran, SQLPrepare</td>
</tr>
<tr>
<td>42S12</td>
<td>Index not found</td>
<td>SQLEndTran, SQLPrepare</td>
</tr>
<tr>
<td>42S21</td>
<td>Column already exists</td>
<td>SQLEndTran, SQLPrepare</td>
</tr>
<tr>
<td>42S22</td>
<td>Column not found</td>
<td>SQLEndTran, SQLPrepare</td>
</tr>
<tr>
<td>44000</td>
<td>WITH CHECK OPTION violation</td>
<td>SQLEndTran, SQLEexecDirect, SQLExecute, SQLParamData, SQLSetPos</td>
</tr>
<tr>
<td>HY000</td>
<td>General error</td>
<td>All ODBC functions except:</td>
</tr>
<tr>
<td>-------</td>
<td>---------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLError</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLError</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetDiagField</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetDiagRec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HY001</th>
<th>Memory allocation error</th>
<th>All ODBC functions except:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SQLError</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLError</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetDiagField</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetDiagRec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HY003</th>
<th>Invalid application buffer type</th>
<th>SQLBindCol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SQLBindParameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetData</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HY004</th>
<th>Invalid SQL data type</th>
<th>SQLBindParameter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SQLGetTypeInfo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HY007</th>
<th>Associated statement is not prepared</th>
<th>SQLCopyDesc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SQLGetDescField</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetDescRec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HY008</th>
<th>Operation canceled</th>
<th>All ODBC functions that can be processed asynchronously:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLBulkOperations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLColAttribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLColumnPrivileges</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLColumns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDescribeCol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDescribeParam</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDisconnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLEndTran</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLExecDirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLExecute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLForeignKeys</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLError</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetDiagField</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetDiagRec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetTypeInfo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLMoreResults</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLNumParams</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLNumResultCols</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLParamData</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLPrepare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLPrimaryKeys</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLProcedureColumns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLStatements</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Functions</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
<td>SQLAllocHandle, SQLBindParameter, SQLBulkOperations, SQLColumnPrivileges, SQLColumns, SQLExecDirect, SQLForeignKeys, SQLGetCursorName, SQLGetData, SQLGetFunctions, SQLNativeSql, SQLPrepare, SQLPrimaryKeys, SQLProcedureColumns, SQLProcedures, SQLPutData, SQLSetConnectAttr, SQLSetCursorName, SQLSetEnvAttr, SQLSetStmtAttr, SQLSpecialColumns, SQLStatistics, SQLTablePrivileges, SQLTables</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
<td>SQLAllocHandle, SQLBindCol, SQLBindParameter, SQLBulkOperations, SQLCloseCursor, SQLColAttribute, SQLColumnPrivileges, SQLColumns, SQLCopyDesc, SQLDescribeCol, SQLDescribeParam, SQLDisconnect</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>ODBC Functions Excluded</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>HY011</td>
<td>Attribute cannot be set now</td>
<td>SQLBulkOperations, SQLParamData, QLSearStmtPos, SQLSetStmtAttr</td>
</tr>
<tr>
<td>HY012</td>
<td>Invalid transaction operation code</td>
<td>SQLEndTran</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
<td>All ODBC functions except:</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
<td>Associated Functions</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>HY014</td>
<td>Limit on the number of handles exceeded</td>
<td>SQLAllocHandle</td>
</tr>
<tr>
<td>HY015</td>
<td>No cursor name available</td>
<td>SQLGetCursorName</td>
</tr>
<tr>
<td>HY016</td>
<td>Cannot modify an implementation row descriptor</td>
<td>SQLCopyDesc, SQLSetDescField, SQLSetDescRec</td>
</tr>
<tr>
<td>HY017</td>
<td>Invalid use of an automatically allocated descriptor handle</td>
<td>SQLFreeHandle, SQLSetStmtAttr</td>
</tr>
<tr>
<td>HY018</td>
<td>Server declined cancel request</td>
<td>SQLCancel</td>
</tr>
<tr>
<td>HY019</td>
<td>Non-character and non-binary data sent in pieces</td>
<td>SQLPutData</td>
</tr>
<tr>
<td>HY020</td>
<td>Attempt to concatenate a null value</td>
<td>SQLPutData</td>
</tr>
<tr>
<td>HY021</td>
<td>Inconsistent descriptor information</td>
<td>SQLBindParameter, SQLCopyDesc, SQLGetDescField, SQLSetDescField, SQLSetDescRec</td>
</tr>
<tr>
<td>HY024</td>
<td>Invalid attribute value</td>
<td>SQLSetConnectAttr, SQLSetEnvAttr, SQLSetStmtAttr</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string or buffer length</td>
<td>SQLBindCol, SQLBindParameter, SQLBrowseConnect, SQLBulkOperations, SQLColAttribute, SQLColumnPrivileges, SQLColumns, SQLConnect, SQLDataSources, SQLDescribeCol, SQLDriverConnect, SQLDrivers, SQLExecDirect, SQLExecute, SQLFetch, SQLFetchScroll, SQLForeignKeys, SQLGetConnectAttr, SQLGetCursorName, SQLGetData, SQLGetDescField, SQLGetInfo, SQLGetStmtAttr</td>
</tr>
<tr>
<td>HY091</td>
<td>Invalid descriptor field identifier</td>
<td>SQLColAttribute</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute/option identifier</td>
<td>SQLAllocHandle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetConnectAttr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLSetConnectAttr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLSetPos</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLSetStmtAttr</td>
</tr>
<tr>
<td>HY095</td>
<td>Function type out of range</td>
<td>SQLGetFunctions</td>
</tr>
<tr>
<td>HY096</td>
<td>Invalid information type</td>
<td>SQLGetInfo</td>
</tr>
<tr>
<td>HY097</td>
<td>Column type out of range</td>
<td>SQLSpecialColumns</td>
</tr>
<tr>
<td>HY098</td>
<td>Scope type out of range</td>
<td>SQLSpecialColumns</td>
</tr>
<tr>
<td>HY099</td>
<td>Nullable type out of range</td>
<td>SQLSpecialColumns</td>
</tr>
<tr>
<td>HY100</td>
<td>Uniqueness option type out of range</td>
<td>SQLStatistics</td>
</tr>
<tr>
<td>HY101</td>
<td>Accuracy option type out of range</td>
<td>SQLStatistics</td>
</tr>
</tbody>
</table>
| HY103 | Invalid retrieval code | SQLDataSources
SQLDrivers |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>HY104</td>
<td>Invalid precision or scale value</td>
<td>SQLBindParameter</td>
</tr>
</tbody>
</table>
| HY105 | Invalid parameter type | SQLBindParameter
SQLExecDirect
SQLExecute
SQLParamData
SQLSetDescField |
| HY106 | Fetch type out of range | SQLExtendedFetch
SQLFetchScroll |
| HY107 | Row value out of range | SQLExtendedFetch
SQLFetch
SQLFetchScroll
SQLSetPos |
| HY109 | Invalid cursor position | SQLExecDirect
SQLExecute
SQLGetData
SQLGetStmtAttr
SQLNativeSql
SQLParamData
SQLSetPos |
| HY110 | Invalid driver completion | SQLDriverConnect |
| HY111 | Invalid bookmark value | SQLExtendedFetch
SQLFetchScroll |
| HYC00 | Optional feature not implemented | SQLBindCol
SQLBindParameter
SQLBulkOperations
SQLColAttribute
SQLColumnPrivileges
SQLColumns
SQLDriverConnect
SQLEndTran
SQLExecDirect
SQLExecute
SQLExtendedFetch
SQLFetch
SQLFetchScroll
SQLForeignKeys
SQLGetConnectAttr
SQLGetData
SQLGetEnvAttr
SQLGetInfo
SQLGetStmtAttr |
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Functions Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>HYT00</td>
<td>Timeout expired</td>
<td>SQLBrowseConnect, SQLBulkOperations, SQLColumnPrivileges, SQLColumns, SQLConnect,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDriverConnect, SQLExecDirect, SQLExecute, SLEXtendedFetch, SQLForeignKeys,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetTypeInfo, SQLParamData, SQLPrepare, SQLPrimaryKeys, SQLProcedureColumns,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLProcedures, SQLSetPos, SQLSpecialColumns, SQLStatistics, SQLTablePrivileges,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLTables</td>
</tr>
<tr>
<td>HYT01</td>
<td>Connection timeout expired</td>
<td>All ODBC functions except:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDrivers, SQLDataSources, SQLGetEnvAttr, SQLSetEnvAttr</td>
</tr>
<tr>
<td>IM001</td>
<td>Driver does not support this function</td>
<td>All ODBC functions except:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLAllocHandle, SQLDataSources, SQLGetTypeInfo, SQLParamData, SQLPrepare,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLPrimaryKeys, SQLProcedureColumns, SQLProcedures, SQLSetPos, SQLSpecialColumns,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLStatistics, SQLTablePrivileges, SQLTables</td>
</tr>
</tbody>
</table>
| IM002 | Data source name not found and no default driver specified | SQLBrowseConnect  
SQLConnect  
SQLDriverConnect |
| IM003 | Specified driver could not be loaded | SQLBrowseConnect  
SQLConnect  
SQLDriverConnect |
| IM004 | Driver’s `SQLAllocHandle` on SQL_HANDLE_ENV failed | SQLBrowseConnect  
SQLConnect  
SQLDriverConnect |
| IM005 | Driver’s `SQLAllocHandle` on SQL_HANDLE_DBC failed | SQLBrowseConnect  
SQLConnect  
SQLDriverConnect |
| IM006 | Driver’s `SQLSetConnectAttr` failed | SQLBrowseConnect  
SQLConnect  
SQLDriverConnect |
| IM007 | No data source or driver specified; dialog prohibited | SQLDriverConnect |
| IM008 | Dialog failed | SQLDriverConnect |
| IM009 | Unable to load translation DLL | SQLBrowseConnect  
SQLConnect  
SQLDriverConnect  
SQLSetConnectAttr |
| IM010 | Data source name too long | SQLBrowseConnect  
SQLConnect  
SQLDriverConnect |
| IM011 | Driver name too long | SQLBrowseConnect  
SQLDriverConnect |
| IM012 | DRIVER keyword syntax error | SQLBrowseConnect  
SQLDriverConnect |
| IM013 | Trace file error | All ODBC functions. |
| IM014 | Invalid name of File DSN | SQLDriverConnect |
| IM015 | Corrupt file data source | SQLDriverConnect |

**Appendix B: ODBC State Transition Tables**

The tables in this appendix show how ODBC functions cause transitions of the environment, connection, statement, and descriptor states. The state of the environment, connection, statement, or descriptor usually dictates when functions that use the corresponding type of handle (environment, connection, statement, or descriptor) can be called. The environment, connection, statement, and descriptor states overlap roughly as shown in the following illustrations. For example, the exact overlap of connection states C5 and C6 and statement states S1 through S12 is data source–dependent, since transactions begin at different times on different data sources, and descriptor state D1e (explicitly allocated descriptor) depends on the state of the statement with which the descriptor is associated, while state D1i (implicitly allocated descriptor) is independent of the state of any statement. For a description of each state, see Environment Transitions, Connection Transitions, Statement Transitions, and Descriptor Transitions, later in this appendix.
The environment and connection states overlap as follows:

\[
\begin{array}{cccccc}
E0 & E1 & E2 \\
C0 & C1 & C2 & C3 & C4 & C5 & C6
\end{array}
\]

The connection and statement states overlap as follows:

\[
\begin{array}{cccccc}
C4 & C5 & C6 \\
S0 & S1 & S2 & S3 & S4 & S5 & S6 & S7 & S8 & S9 & S10 & S11 & S12
\end{array}
\]

The statement and descriptor states overlap as follows:

\[
\begin{array}{cccccc}
S0 & S1 & S2 & S3 & S4 & S5 & S6 & S7 & S8 & S9 & S10 & S11 & S12 \\
D0 & D1a & D1b & D1c & D1d & D1e & D1f & D1g & D1h & D1i & D1j & D1k & D1l
\end{array}
\]

The connection and descriptor states overlap as follows:

\[
\begin{array}{cccccc}
C4 & C5 & C6 \\
D0 & D1a & D1b & D1c & D1d & D1e & D1f & D1g & D1h & D1i & D1j & D1k & D1l
\end{array}
\]

Each entry in a transition table can be one of the following values:

- \(E\) — The state is unchanged after executing the function.
- \(E, C, S, D\) — The environment, connection, statement, or descriptor state moves to the specified state.
- \(IH\) — An invalid handle was passed to the function. If the handle was a null handle or was a valid handle of the wrong type — for example, a
  connection handle was passed when a statement handle was required — the function returns SQL_INVALID_HANDLE; otherwise the behavior is
  undefined and probably fatal. This error is shown only when it is the only possible outcome of calling the function in the specified state. This error does
  not change the state and is always detected by the Driver Manager, as indicated by the parentheses.
- \(NS\) — Next State. The statement transition is the same as if the statement had not gone through the asynchronous states. For example, suppose a
  statement that creates a result set enters state S11 from state S1 because SQLExecDirect returned SQL_STILL_EXECUTING. The NS notation in state
  S11 means that the transitions for the state are the same as those for a statement in state S1 that creates a result set. If SQLExecDirect returns
  an error, the statement remains in state S11; if it succeeds, the statement moves to state S5; if it needs data, the statement moves to state S8; and if it
  is still executing, it remains in state S11.
- \(XXXX\) or \((XXXX)\) — An SQLSTATE that is related to the transition table; SQLSTATEs detected by the Driver Manager are enclosed in parentheses.
  The function returned SQL_ERROR and the specified SQLSTATE, but the state does not change. For example, if SQLExecute is called before
  SQLPrepare, it returns SQLSTATE HY010 (Function sequence error).

**Note**

The tables do not show errors unrelated to the transition tables that do not change the state. For example, when SQLAllocHandle is called in environment
state E1 and returns SQLSTATE HY001 (Memory allocation error), the environment remains in state E1; this is not shown in the environment transition table
for SQLAllocHandle.

If the environment, connection, statement, or descriptor can move to more than one state, each possible state is shown and one or more footnotes explain the
conditions under which each transition takes place. The following footnotes may appear in any table.

<table>
<thead>
<tr>
<th>Footnote</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Before or after. The cursor was positioned before the start of the result set or after the end of the result set.</td>
</tr>
<tr>
<td>c</td>
<td>Current function. The current function was executing asynchronously.</td>
</tr>
<tr>
<td>d</td>
<td>Need data. The function returned SQL_NEED_DATA.</td>
</tr>
<tr>
<td>e</td>
<td>Error. The function returned SQL_ERROR.</td>
</tr>
</tbody>
</table>
| i        | Invalid row. The cursor was positioned on a row in the result set and either the row had been deleted or an error had occurred in an operation
  on the row. If the row status array existed, the value in the row status array for the row was SQL_ROW_DELETED or SQL_ROW_ERROR. (The
  row status array is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute.) |
| nf       | Not found. The function returned SQL_NO_DATA. This does not apply when SQLExecDirect, SQLExecute, or SQLParamData returns
  SQL_NO_DATA after executing a searched update or delete statement. |
np Not prepared. The statement was not prepared.
nr No results. The statement will not or did not create a result set.
o Other function. Another function was executing asynchronously.
p Prepared. The statement was prepared.
r Results. The statement will or did create a (possibly empty) result set.
s Success. The function returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.
v Valid row. The cursor was positioned on a row in the result set, and the row had been successfully inserted, successfully updated, or another operation on the row had been successfully completed. If the row status array existed, the value in the row status array for the row was SQL_ROW_ADDED, SQL_ROW_SUCCESS, or SQL_ROW_UPDATED. (The row status array is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute.)
x Executing. The function returned SQL_STILL_EXECUTING.

SQLFreeHandle

In this example, the row in the environment state transition table for SQLFreeHandle when HandleType is SQL_HANDLE_ENV is as follows.

<table>
<thead>
<tr>
<th>E0</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Connection</td>
</tr>
<tr>
<td>(IH)</td>
<td>E0</td>
<td>(HY010)</td>
</tr>
</tbody>
</table>

If SQLFreeHandle is called in environment state E0 with HandleType set to SQL_HANDLE_ENV, the Driver Manager returns SQL_INVALID_HANDLE. If it is called in state E1 with HandleType set to SQL_HANDLE_ENV, the environment moves to state E0 if the function succeeds and remains in state E1 if the function fails. If it is called in state E2 with HandleType set to SQL_HANDLE_ENV, the Driver Manager always returns SQL_ERROR and SQLSTATE HY010 (Function sequence error) and the environment remains in state E2.

To understand the state transition tables, it is necessary to understand which item (environment, connection, statement, or descriptor) they refer to. Suppose a function accepts the handle of an item of type X. The X state transition table for that function describes how calling the function, with the handle of an item of type X, affects that item. For example, SQLDisconnect accepts a connection handle. The connection state transition table for SQLDisconnect describes how SQLDisconnect affects the state of the connection for which it is called.

Suppose a function accepts the handle of an item of type Y, where Y is not equal to X. The X state transition table for that function describes how calling the function, with a handle of type X that is associated with the item of type Y, affects the item of type Y. For example, the statement state transition table for SQLDisconnect describes how SQLDisconnect affects the state of a statement when called with the handle of the connection with which the statement is associated.

This appendix contains the following topics.

- Environment Transitions
- Connection Transitions
- Statement Transitions
- Descriptor Transitions

© 2016 Microsoft

Environment Transitions

ODBC environments have the following three states.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>Unallocated environment</td>
</tr>
<tr>
<td>E1</td>
<td>Allocated environment, unallocated connection</td>
</tr>
<tr>
<td>E2</td>
<td>Allocated environment, allocated connection</td>
</tr>
</tbody>
</table>
The following tables show how each ODBC function affects the environment state.

### SQLAllocHandle

<table>
<thead>
<tr>
<th>E0 Unallocated</th>
<th>E1 Allocated</th>
<th>E2 Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0(1]</td>
<td>--[4]</td>
<td>--[4]</td>
</tr>
<tr>
<td>(IH)(3]</td>
<td>(IH)</td>
<td>--[4]</td>
</tr>
</tbody>
</table>

[1] This row shows transitions when HandleType was SQL_HANDLE_ENV.
[2] This row shows transitions when HandleType was SQL_HANDLE_DBC.
[3] This row shows transitions when HandleType was SQL_HANDLE_STMT or SQL_HANDLE_DESC.
[4] Calling `SQLAllocHandle` with `OutputHandlePtr` pointing to a valid handle overwrites that handle. This might be an application programming error.
[5] The SQL_ATTR_ODBC_VERSION environment attribute had been set on the environment.
[6] The SQL_ATTR_ODBC_VERSION environment attribute had not been set on the environment.

### SQLDataSources and SQLDrivers

<table>
<thead>
<tr>
<th>E0 Unallocated</th>
<th>E1 Allocated</th>
<th>E2 Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td>--[1] (HY010)[2]</td>
<td>--[1] (HY010)[2]</td>
</tr>
</tbody>
</table>

[1] The SQL_ATTR_ODBC_VERSION environment attribute had been set on the environment.
[2] The SQL_ATTR_ODBC_VERSION environment attribute had not been set on the environment.

### SQLEndTran

<table>
<thead>
<tr>
<th>E0 Unallocated</th>
<th>E1 Allocated</th>
<th>E2 Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)(2]</td>
<td>(IH)</td>
<td>--</td>
</tr>
</tbody>
</table>

[1] This row shows transitions when HandleType was SQL_HANDLE_ENV.
[2] This row shows transitions when HandleType was SQL_HANDLE_DBC.
[3] The SQL_ATTR_ODBC_VERSION environment attribute had been set on the environment.
[4] The SQL_ATTR_ODBC_VERSION environment attribute had not been set on the environment.

### SQLFreeHandle

<table>
<thead>
<tr>
<th>E0 Unallocated</th>
<th>E1 Allocated</th>
<th>E2 Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)(1]</td>
<td>E0</td>
<td>(HY010)</td>
</tr>
<tr>
<td>(IH)(3]</td>
<td>(IH)</td>
<td>--</td>
</tr>
</tbody>
</table>
This row shows transitions when `HandleType` was SQL_HANDLE_ENV.

This row shows transitions when `HandleType` was SQL_HANDLE_DBC.

This row shows transitions when `HandleType` was SQL_HANDLE_STMT or SQL_HANDLE_DESC.

There were other allocated connection handles.

The connection handle specified in `Handle` was the only allocated connection handle.

**SQLGetDiagField and SQLGetDiagRec**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>E1</td>
<td>E2</td>
</tr>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Connection</td>
</tr>
<tr>
<td>(IH)[1]</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>(IH)[2]</td>
<td>(IH)</td>
<td>--</td>
</tr>
</tbody>
</table>

This row shows transitions when `HandleType` was SQL_HANDLE_ENV.

This row shows transitions when `HandleType` was SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC.

**SQLGetEnvAttr**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>E1</td>
<td>E2</td>
</tr>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Connection</td>
</tr>
<tr>
<td>(IH)</td>
<td>--[1] (HY010)[2]</td>
<td>--</td>
</tr>
</tbody>
</table>

The SQL_ATTR_ODBC_VERSION environment attribute had been set on the environment.

The SQL_ATTR_ODBC_VERSION environment attribute had not been set on the environment.

**SQLSetEnvAttr**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>E1</td>
<td>E2</td>
</tr>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Connection</td>
</tr>
<tr>
<td>(IH)</td>
<td>--[1] (HY010)[2]</td>
<td>(HY011)</td>
</tr>
</tbody>
</table>

The SQL_ATTR_ODBC_VERSION environment attribute had been set on the environment.

The Attribute argument was not SQL_ATTR_ODBC_VERSION, and the SQL_ATTR_ODBC_VERSION environment attribute had not been set on the environment.

**All Other ODBC Functions**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E0</td>
<td>E1</td>
<td>E2</td>
</tr>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Connection</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>--</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**Connection Transitions**

ODBC connections have the following states.
The following tables show how each ODBC function affects the connection state.

### SQLAllocHandle

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>Unallocated environment, unallocated connection</td>
</tr>
<tr>
<td>C1</td>
<td>Allocated environment, unallocated connection</td>
</tr>
<tr>
<td>C2</td>
<td>Allocated environment, allocated connection</td>
</tr>
<tr>
<td>C3</td>
<td>Connection function needs data</td>
</tr>
<tr>
<td>C4</td>
<td>Connected connection</td>
</tr>
<tr>
<td>C5</td>
<td>Connected connection, allocated statement</td>
</tr>
<tr>
<td>C6</td>
<td>Connected connection, transaction in progress. It is possible for a connection to be in state C6 with no statements allocated on the connection. For example, suppose the connection is in manual commit mode and is in state C4. If a statement is allocated, executed (starting a transaction), and then freed, the transaction remains active but there are no statements on the connection.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQLAllocHandle</th>
<th>C0</th>
<th>C1 Unallocated</th>
<th>C2 Allocated</th>
<th>C3 Need Data</th>
<th>C4 Connected</th>
<th>C5 Statement</th>
<th>C6 Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[1] This row shows transitions when HandleType was SQL_HANDLE_ENV.
[2] This row shows transitions when HandleType was SQL_HANDLE_DBC.
[3] This row shows transitions when HandleType was SQL_HANDLE_STMT.
[4] This row shows transitions when HandleType was SQL_HANDLE_DESC.
[5] Calling SQLAllocHandle with OutputHandlePtr pointing to a valid handle overwrites that handle without regard for the previous contents of that handle, and might cause problems for ODBC drivers. It is incorrect ODBC application programming to call SQLAllocHandle twice with the same application variable defined for *OutputHandlePtr without calling SQLFreeHandle to free the handle before reallocating it. Overwriting ODBC handles in such a manner can lead to inconsistent behavior or errors on the part of ODBC drivers.

### SQLBrowseConnect

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>No Env.</td>
</tr>
<tr>
<td>C1</td>
<td>Unallocated</td>
</tr>
<tr>
<td>C2</td>
<td>Allocated</td>
</tr>
<tr>
<td>C3</td>
<td>Need Data</td>
</tr>
<tr>
<td>C4</td>
<td>Connected</td>
</tr>
<tr>
<td>C5</td>
<td>Statement</td>
</tr>
<tr>
<td>C6</td>
<td>Transaction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQLBrowseConnect</th>
<th>C0</th>
<th>C1 Unallocated</th>
<th>C2 Allocated</th>
<th>C3 Need Data</th>
<th>C4 Connected</th>
<th>C5 Statement</th>
<th>C6 Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SQLCloseCursor

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>No Env.</td>
</tr>
<tr>
<td>C1</td>
<td>Unallocated</td>
</tr>
<tr>
<td>C2</td>
<td>Allocated</td>
</tr>
<tr>
<td>C3</td>
<td>Need Data</td>
</tr>
<tr>
<td>C4</td>
<td>Connected</td>
</tr>
<tr>
<td>C5</td>
<td>Statement</td>
</tr>
<tr>
<td>C6</td>
<td>Transaction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQLCloseCursor</th>
<th>C0</th>
<th>C1 Unallocated</th>
<th>C2 Allocated</th>
<th>C3 Need Data</th>
<th>C4 Connected</th>
<th>C5 Statement</th>
<th>C6 Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[1] The connection was in manual-commit mode.
[2] The connection was in auto-commit mode.
SQLColumnPrivileges, SQLColumns, SQLForeignKeys, SQLGetTypeInfo, SQLPrimaryKeys, SQLProcedureColumns, SQLProcedures, SQLSpecialColumns, SQLStatistics, SQLTablePrivileges, and SQLTables

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
</tbody>
</table>

[1] The connection was in auto-commit mode, or the data source did not begin a transaction.
[2] The connection was in manual-commit mode, and the data source began a transaction.

**SQLConnect**

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>C4</td>
<td>(08002)</td>
<td>(08002)</td>
<td>(08002)</td>
<td>(08002)</td>
</tr>
</tbody>
</table>

**SQLCopyDesc, SQLGetDescField, SQLGetDescRec, SQLSetDescField, and SQLSetDescRec**

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>--[1]</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

[1] In this state, the only descriptors available to the application are explicitly allocated descriptors.

**SQLDataSources and SQLDrivers**

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**SQLDisconnect**

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>(08003)</td>
<td>C2</td>
<td>C2</td>
<td>C2</td>
<td>25000</td>
</tr>
</tbody>
</table>

**SQLDriverConnect**

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>C4 s -- n[T]</td>
<td>(08002)</td>
<td>(08002)</td>
<td>(08002)</td>
<td>(08002)</td>
</tr>
</tbody>
</table>
### SQLEndTran

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)[2]</td>
<td>(IH)</td>
<td>(08003)</td>
<td>(08003)</td>
<td>--</td>
<td>--</td>
<td>C5</td>
</tr>
</tbody>
</table>

1. This row shows transitions when `HandleType` was SQL_HANDLE_ENV.
2. This row shows transitions when `HandleType` was SQL_HANDLE_DBC.
3. Because the connection is not in a connected state, it is unaffected by the transaction.
4. The commit or rollback failed on the connection. The function returns SQL_ERROR in this case.
5. The commit or rollback succeeded on the connection. The function returns SQL_ERROR if the commit or rollback failed on another connection, or the function returns SQL_SUCCESS if the commit or rollback succeeded on all connections.
6. There was at least one statement allocated on the connection.
7. There were no statements allocated on the connection.
8. The connection had at least one statement for which there was an open cursor, and the data source preserves cursors when transactions are committed or rolled back, whichever applies (depending on whether `CompletionType` was SQL_COMMIT or SQL_ROLLBACK). For more information, see the SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR attributes in SQLGetInfo.
9. If the connection had any statements for which there were open cursors, the cursors were not preserved when the transaction was committed or rolled back.

### SQLExecDirect and SQLExecute

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
</tbody>
</table>

1. The connection was in auto-commit mode, and the statement executed was not a cursor specification (such as a SELECT statement); or the connection was in manual-commit mode, and the statement executed did not begin a transaction.
2. The connection was in auto-commit mode, and the statement executed was a cursor specification (such as a SELECT statement).
3. The connection was in manual-commit mode, and the data source began a transaction.

### SQLFreeHandle

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)[1]</td>
<td>C0</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(HY010)</td>
</tr>
<tr>
<td>(IH)[2]</td>
<td>(IH)</td>
<td>(C1)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(HY010)</td>
</tr>
<tr>
<td>(IH)[4]</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

1. This row shows transitions when `HandleType` was SQL_HANDLE_ENV.
2. This row shows transitions when `HandleType` was SQL_HANDLE_DBC.
3. This row shows transitions when `HandleType` was SQL_HANDLE_STMT.
4. This row shows transitions when `HandleType` was SQL_HANDLE_DESC.
5. There was only one statement allocated on the connection.
6. There were multiple statements allocated on the connection.
7. The connection was in manual-commit mode.
The connection was in auto-commit mode.

SQLFreeStmt

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
</tr>
<tr>
<td>(IH)[2]</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

[1] This row shows transactions when the Option argument is SQL_CLOSE.

[2] This row shows transactions when the Option argument is SQL_UNBIND or SQL_RESET_PARAMS.

[3] The connection was in auto-commit mode, and no cursors were open on any statements except this one.

[4] The connection was in manual-commit mode, or it was in auto-commit mode and a cursor was open on at least one other statement.

SQLGetConnectAttr

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>--[1] (08003)[2]</td>
<td>(HY010)</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

[1] The Attribute argument was SQL_ATTR_ACCESS_MODE, SQL_ATTR_AUTOCOMMIT, SQL_ATTR_LOGIN_TIMEOUT, SQL_ATTR_ODBC_CURSORS, SQL_ATTR_TRACE, or SQL_ATTR_TRACEFILE, or a value had been set for the connection attribute.

[2] The Attribute argument was not SQL_ATTR_ACCESS_MODE, SQL_ATTR_AUTOCOMMIT, SQL_ATTR_LOGIN_TIMEOUT, SQL_ATTR_ODBC_CURSORS, SQL_ATTR_TRACE, or SQL_ATTR_TRACEFILE, and a value had not been set for the connection attribute.

SQLGetDiagField and SQLGetDiagRec

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
</tr>
<tr>
<td>(IH)[1]</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>(IH)[2]</td>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>(IH)[3]</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>(IH)[4]</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

[1] This row shows transitions when HandleType was SQL_HANDLE_ENV.

[2] This row shows transitions when HandleType was SQL_HANDLE_DBC.

[3] This row shows transitions when HandleType was SQL_HANDLE_STMT.

[4] This row shows transitions when HandleType was SQL_HANDLE_DESC.

SQLGetEnvAttr

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
</tr>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
### SQLGetFunctions

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

### SQLGetInfo

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

[1] The InfoType argument was SQL_ODBC_VER.

[2] The InfoType argument was not SQL_ODBC_VER.

### SQLMoreResults

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

[1] The connection was in auto-commit mode, and the call to SQLMoreResults has not initialized the processing of a result set of a cursor specification.

[2] The connection was in auto-commit mode, and the call to SQLMoreResults has initialized the processing of a result set of a cursor specification.

[3] The connection was in manual-commit mode.

### SQLNativeSql

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>(08003)</td>
<td>(08003)</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

### SQLPrepare

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

[1] The connection was in auto-commit mode, or the data source did not begin a transaction.

[2] The connection was in manual-commit mode, and the data source began a transaction.

### SQLSetConnectAttr

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>(HY010)</td>
<td>(08002)[4]</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

The Attribute argument was not SQL_ATTR_TRANSLATE_LIB or SQL_ATTR_TRANSLATE_OPTION.

The Attribute argument was SQL_ATTR_TRANSLATE_LIB or SQL_ATTR_TRANSLATE_OPTION.

The Attribute argument was not SQL_ATTR_ODBC_CURSORS or SQL_ATTR_PACKET_SIZE.

The Attribute argument was SQL_ATTR_ODBC_CURSORS.

The Attribute argument was SQL_ATTR_PACKET_SIZE.

The Attribute argument was not SQL_ATTR_AUTOCOMMIT, or the Attribute argument was SQL_ATTR_AUTOCOMMIT and setting this attribute did not commit the transaction.

The Attribute argument was SQL_ATTR_TXN_ISOLATION.

The Attribute argument was SQL_ATTR_AUTOCOMMIT, and setting this attribute committed the transaction.

**SQLSetEnvAttr**

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**All Other ODBC Functions**

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Env.</td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Need Data</td>
<td>Connected</td>
<td>Statement</td>
<td>Transaction</td>
</tr>
<tr>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**Statement Transitions**

ODBC statements have the following states.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>Unallocated statement. (The connection state must be C4, C5, or C6. For more information, see <a href="#">Connection Transitions</a>.)</td>
</tr>
<tr>
<td>S1</td>
<td>Allocated statement.</td>
</tr>
<tr>
<td>S2</td>
<td>Prepared statement. No result set will be created.</td>
</tr>
<tr>
<td>S3</td>
<td>Prepared statement. A (possibly empty) result set will be created.</td>
</tr>
<tr>
<td>S4</td>
<td>Statement executed and no result set was created.</td>
</tr>
<tr>
<td>S5</td>
<td>Statement executed and a (possibly empty) result set was created. The cursor is open and positioned before the first row of the result set.</td>
</tr>
<tr>
<td>S6</td>
<td>Cursor positioned with <code>SQLFetch</code> or <code>SQLFetchScroll</code>.</td>
</tr>
<tr>
<td>S7</td>
<td>Cursor positioned with <code>SQLExtendedFetch</code>.</td>
</tr>
<tr>
<td>S8</td>
<td>Function needs data. <code>SQLParamData</code> has not been called.</td>
</tr>
<tr>
<td>S9</td>
<td>Function needs data. <code>SQLPutData</code> has not been called.</td>
</tr>
<tr>
<td>S10</td>
<td>Function needs data. <code>SQLPutData</code> has been called.</td>
</tr>
<tr>
<td>S11</td>
<td>Still executing. A statement is left in this state after a function that is executed asynchronously returns <code>SQL_STILL_EXECUTING</code>. A statement is</td>
</tr>
</tbody>
</table>
temporarily in this state while any function that accepts a statement handle is executing. Temporary residence in state S11 is not shown in any
state tables except the state table for SQLCancel. While a statement is temporarily in state S11, the function can be canceled by calling
SQLCancel from another thread.

S12
Asynchronous execution canceled. In S12, an application must call the canceled function until it returns a value other than SQL_STILL_EXECUTING.
The function was canceled successfully only if the function returns SQL_ERROR and SQLSTATE HY008 (Operation canceled). If it returns any other
value, such as SQL_SUCCESS, the cancel operation failed and the function executed normally.

States S2 and S3 are known as the prepared states, states S5 through S7 as the cursor states, states S8 through S10 as the need data states, and states S11
and S12 as the asynchronous states. In each of these groups, the transitions are shown separately only when they are different for each state in the group; in
most cases, the transitions for each state in each a group are the same.

The following tables show how each ODBC function affects the statement state.

**SQLAllocHandle**

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td></td>
<td>Allocated</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
<td>(HY010)</td>
</tr>
</tbody>
</table>

[1] This row shows transitions when HandleType was SQL_HANDLE_ENV.
[2] This row shows transitions when HandleType was SQL_HANDLE_DBC.
[3] This row shows transitions when HandleType was SQL_HANDLE_STMT.
[4] This row shows transitions when HandleType was SQL_HANDLE_DESC.
[5] Calling SQLAllocHandle with OutputHandlePtr pointing to a valid handle overwrites that handle without regard for the previous contents to that
handle and might cause problems for ODBC drivers. It is incorrect ODBC application programming to call SQLAllocHandle twice with the same application
variable defined for *OutputHandlePtr without calling SQLFreeHandle to free the handle before reallocating it. Overwriting ODBC handles in such a manner
might lead to inconsistent behavior or errors on the part of ODBC drivers.

**SQLBindCol**

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td></td>
<td>Allocated</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
<td>(HY010)</td>
</tr>
</tbody>
</table>

**SQLBindParameter**

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td></td>
<td>Allocated</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
<td>(HY010)</td>
</tr>
</tbody>
</table>

**SQLBrowseConnect, SQLConnect, and SQLDriverConnect**

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td></td>
<td>Allocated</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(08002)</td>
<td>(08002)</td>
<td>(08002)</td>
<td>(08002)</td>
<td>(08002)</td>
<td>(08002)</td>
<td>(08002)</td>
</tr>
</tbody>
</table>
### SQLBulkOperations

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>Unallocated</td>
</tr>
<tr>
<td>S1</td>
<td>Allocated</td>
</tr>
<tr>
<td>S2–S3</td>
<td>Prepared</td>
</tr>
<tr>
<td>S4</td>
<td>Executed</td>
</tr>
<tr>
<td>S5–S7</td>
<td>Cursor</td>
</tr>
<tr>
<td>S8–S10</td>
<td>Need Data</td>
</tr>
<tr>
<td>S11–S12</td>
<td>Async</td>
</tr>
</tbody>
</table>

- (IH) 24000 See next table
- (HY010) 24000

#### SQLBulkOperations (Cursor States)

<table>
<thead>
<tr>
<th>S5</th>
<th>Opened</th>
<th>S6</th>
<th>SQLFetch or SQLFetchScroll</th>
<th>S7</th>
<th>SQLExtendedFetch</th>
</tr>
</thead>
</table>

#### SQLCancel

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>Unallocated</td>
</tr>
<tr>
<td>S1</td>
<td>Allocated</td>
</tr>
<tr>
<td>S2–S3</td>
<td>Prepared</td>
</tr>
<tr>
<td>S4</td>
<td>Executed</td>
</tr>
<tr>
<td>S5–S7</td>
<td>Cursor</td>
</tr>
<tr>
<td>S8–S10</td>
<td>Need Data</td>
</tr>
<tr>
<td>S11–S12</td>
<td>Async</td>
</tr>
</tbody>
</table>


1. SQLExecDirect returned SQL_NEED_DATA.
2. SQLExecute returned SQL_NEED_DATA.
3. SQLBulkOperations returned SQL_NEED_DATA.
4. SQLSetPos returned SQL_NEED_DATA.
5. SQLFetch, SQLFetchScroll, or SQLExtendedFetch had not been called.
6. SQLFetch or SQLFetchScroll had been called.
7. SQLExtendedFetch had been called.

#### SQLCancel (Asynchronous States)

<table>
<thead>
<tr>
<th>S11</th>
<th>Still executing</th>
</tr>
</thead>
<tbody>
<tr>
<td>S12</td>
<td>Asynch canceled</td>
</tr>
</tbody>
</table>


1. The statement was temporarily in state S11 while a function was executing. SQLCancel was called from a different thread.
2. The statement was in state S11 because a function called asynchronously returned SQL_STILL_EXECUTING.

#### SQLCloseCursor

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>Unallocated</td>
</tr>
<tr>
<td>S1</td>
<td>Allocated</td>
</tr>
<tr>
<td>S2–S3</td>
<td>Prepared</td>
</tr>
<tr>
<td>S4</td>
<td>Executed</td>
</tr>
<tr>
<td>S5–S7</td>
<td>Cursor</td>
</tr>
<tr>
<td>S8–S10</td>
<td>Need Data</td>
</tr>
<tr>
<td>S11–S12</td>
<td>Async</td>
</tr>
</tbody>
</table>

- (IH) 24000 24000 24000 S1 [np] S3 [p] (HY010) (HY010)

#### SQLColAttribute

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
</tr>
<tr>
<td>S2–S3</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td></td>
</tr>
<tr>
<td>S5–S7</td>
<td></td>
</tr>
<tr>
<td>S8–S10</td>
<td></td>
</tr>
<tr>
<td>S11–S12</td>
<td></td>
</tr>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
</tr>
</tbody>
</table>

### SQLColAttribute (Prepared States)

<table>
<thead>
<tr>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Results</td>
<td>Results</td>
</tr>
</tbody>
</table>

[1] FieldIdentifier was SQL_DESC_COUNT.

[2] FieldIdentifier was not SQL_DESC_COUNT.

### SQLColumnPrivileges, SQLColumns, SQLForeignKeys, SQLGetTypeInfo, SQLPrimaryKeys, SQLProcedureColumns, SQLProcedures, SQLSpecialColumns, SQLStatistics, SQLTablePrivileges, and SQLTables

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
</tbody>
</table>

[1] The current result is the last or only result, or there are no current results. For more information about multiple results, see Multiple Results.

[2] The current result is not the last result.

### SQLColumnPrivileges, SQLColumns, SQLForeignKeys, SQLGetTypeInfo, SQLPrimaryKeys, SQLProcedureColumns, SQLProcedures, SQLSpecialColumns, SQLStatistics, SQLTablePrivileges, and SQLTables (Cursor States)

<table>
<thead>
<tr>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opened</td>
<td>SQLFetch or SQLFetchScroll</td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td>24000</td>
<td>(24000)[1]</td>
<td>(24000)</td>
</tr>
</tbody>
</table>

[1] This error is returned by the Driver Manager if SQLFetch or SQLFetchScroll has not returned SQL_NO_DATA and is returned by the driver if SQLFetch or SQLFetchScroll has returned SQL_NO_DATA.

### SQLCopyDesc

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)[1]</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
<td>NS [c] and [3] (HY010) [o] or [4]</td>
</tr>
<tr>
<td>(IH)[2]</td>
<td>(HY010)</td>
<td>See next table</td>
<td>24000</td>
<td>-- [s] S11 x</td>
<td>(HY010)</td>
<td>NS [c] and [3] (HY010) [o] or [4]</td>
</tr>
</tbody>
</table>

[1] This row shows transitions when the SourceDescHandle argument was an ARD, APD, or IPD.

[2] This row shows transitions when the SourceDescHandle argument was an IRD.

[3] The SourceDescHandle and TargetDescHandle arguments were the same as in the SQLCopyDesc function that is running asynchronously.
Either the `SourceDescHandle` argument or the `TargetDescHandle` argument (or both) were different than in the `SQLCopyDesc` function that is running asynchronously.

**SQLCopyDesc (Prepared States)**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Results</td>
<td>Results</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24000[1]</td>
<td>-- [s] S1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[1] This row shows transitions when the `SourceDescHandle` argument was an IRD.

**SQLDataSources and SQLDrivers**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S1</td>
<td>S2–S3</td>
<td>S4</td>
<td>S5–S7</td>
<td>S8–S10</td>
<td>S11–S12</td>
<td></td>
</tr>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
</tbody>
</table>

**SQLDescribeCol**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S1</td>
<td>S2–S3</td>
<td>S4</td>
<td>S5–S7</td>
<td>S8–S10</td>
<td>S11–S12</td>
</tr>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)</td>
<td>(HYO10)</td>
<td>See next table</td>
<td>24000</td>
<td>-- [s] S1</td>
<td>(HYO10)</td>
<td>NS [c] (HYO10)</td>
</tr>
</tbody>
</table>

**SQLDescribeCol (Prepared States)**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Results</td>
<td>Results</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07005</td>
<td>-- [s] S1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SQLDescribeParam**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S1</td>
<td>S2–S3</td>
<td>S4</td>
<td>S5–S7</td>
<td>S8–S10</td>
<td>S11–S12</td>
</tr>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)</td>
<td>(HYO10)</td>
<td>-- [s] S1</td>
<td>(HYO10)</td>
<td>(HYO10)</td>
<td>(HYO10)</td>
<td>NS [c] (HYO10)</td>
</tr>
</tbody>
</table>

**SQLDisconnect**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S1</td>
<td>S2–S3</td>
<td>S4</td>
<td>S5–S7</td>
<td>S8–S10</td>
</tr>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
</tr>
</tbody>
</table>

[1] Calling `SQLDisconnect` frees all statements associated with the connection. Furthermore, this returns the connection state to C2; the connection state must be C4 before the statement state is S0.
**SQLExecDirect (Cursor States)**

<table>
<thead>
<tr>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opened</td>
<td>SQLFetch or SQLFetchScroll</td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td>24000</td>
<td>(24000) [1]</td>
<td>(24000)</td>
</tr>
</tbody>
</table>

[1] This error is returned by the Driver Manager if SQLFetch or SQLFetchScroll has not returned SQL_NO_DATA, and is returned by the driver if SQLFetch or SQLFetchScroll has returned SQL_NO_DATA.

**SQLExecute**

<table>
<thead>
<tr>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
<td>See next table</td>
<td>S2 [e], p, and [1] S4 [s], [p], [nr], and [1] S5 [s], [p], [r], and [1] S8 [d], [p], and [1] S11 [x], [p], and [1] 24000 [p] and [2] (HY010) [np]</td>
<td>See cursor states table (HY010) [0]</td>
<td>NS [c] (HY010) [o]</td>
<td></td>
</tr>
</tbody>
</table>

[1] The current result is the last or only result, or there are no current results. For more information about multiple results, see Multiple Results.

[2] The current result is not the last result.
### SQLExecute (Prepared States)

<table>
<thead>
<tr>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Results</td>
<td>Results</td>
</tr>
<tr>
<td>S4 [s] S8 [d] S11 [x]</td>
<td>S5 [s] S8 [d] S11 [x]</td>
</tr>
</tbody>
</table>

### SQLExecute (Cursor States)

<table>
<thead>
<tr>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opened</td>
<td>SQLFetch or SQLFetchScroll</td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td>24000 [p] (HY010) [np]</td>
<td>(24000) [p], [1] (HY010) [np]</td>
<td>(24000) [p] (HY010) [np]</td>
</tr>
</tbody>
</table>

[1] This error is returned by the Driver Manager if SQLFetch or SQLFetchScroll has not returned SQL_NO_DATA, and is returned by the driver if SQLFetch or SQLFetchScroll has returned SQL_NO_DATA.

### SQLExtendedFetch

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)</td>
<td>(S1010)</td>
<td>(S1010)</td>
<td>24000</td>
<td>See next table</td>
<td>(S1010)</td>
<td>NS [c] (S1010) [o]</td>
</tr>
</tbody>
</table>

### SQLExtendedFetch (Cursor States)

<table>
<thead>
<tr>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opened</td>
<td>SQLFetch or SQLFetchScroll</td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td>S7 [s] or [nf] S11 [x]</td>
<td>(S1010)</td>
<td>-- [s] or [nf] S11 [x]</td>
</tr>
</tbody>
</table>

### SQLFetch and SQLFetchScroll

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>24000</td>
<td>See next table</td>
<td>(HY010)</td>
<td>NS [c] (HY010) [o]</td>
</tr>
</tbody>
</table>

### SQLFetch and SQLFetchScroll (Cursor states)

<table>
<thead>
<tr>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opened</td>
<td>SQLFetch or SQLFetchScroll</td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td>S6 [s] or [nf] S11 [x]</td>
<td>-- [s] or [nf] S11 [x]</td>
<td>(HY010)</td>
</tr>
</tbody>
</table>

### SQLFreeHandle

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
</table>
### SQLFreeStmt

<table>
<thead>
<tr>
<th>State</th>
<th>Unallocated</th>
<th>Allocated</th>
<th>Prepared</th>
<th>Executed</th>
<th>Cursor</th>
<th>Need Data</th>
<th>Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>S1 [np]</td>
<td>S2 [p]</td>
<td>S1 [np]</td>
<td>S3 [p]</td>
<td>(HY010)</td>
</tr>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
</tr>
</tbody>
</table>

[1] This row shows transitions when `Option` was SQL_CLOSE.

[2] This row shows transitions when `Option` was SQL_UNBIND or SQL_RESET_PARAMS. If the `Option` argument was SQL_DROP and the underlying driver is an ODBC 3.x driver, the Driver Manager maps this to a call to SQLFreeHandle with `HandleType` set to SQL_HANDLE_STMT. For more information, see the transition table for SQLFreeHandle.

### SQLGetConnectAttr

<table>
<thead>
<tr>
<th>State</th>
<th>Unallocated</th>
<th>Allocated</th>
<th>Prepared</th>
<th>Executed</th>
<th>Cursor</th>
<th>Need Data</th>
<th>Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
<td>(HY010)</td>
</tr>
</tbody>
</table>

### SQLGetCursorName

<table>
<thead>
<tr>
<th>State</th>
<th>Unallocated</th>
<th>Allocated</th>
<th>Prepared</th>
<th>Executed</th>
<th>Cursor</th>
<th>Need Data</th>
<th>Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td></td>
</tr>
</tbody>
</table>

### SQLGetData

<table>
<thead>
<tr>
<th>State</th>
<th>Unallocated</th>
<th>Allocated</th>
<th>Prepared</th>
<th>Executed</th>
<th>Cursor</th>
<th>Need Data</th>
<th>Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(24000)</td>
<td>See next table</td>
<td>(HY010)</td>
<td>NS [c] (HY010) [o]</td>
<td></td>
</tr>
</tbody>
</table>

### SQLGetData (Cursor States)

<table>
<thead>
<tr>
<th>State</th>
<th>Unallocated</th>
<th>Allocated</th>
<th>Prepared</th>
<th>Executed</th>
<th>Cursor</th>
<th>Need Data</th>
<th>Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>S5</td>
<td>Opened</td>
<td>SQLFetch or SQLFetchScroll</td>
<td>SQLExtendedFetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### SQLGetDescField and SQLGetDescRec

<table>
<thead>
<tr>
<th></th>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
</table>

[1] The DescriptorHandle argument was an APD or ARD.

[2] The DescriptorHandle argument was an IPD.

[3] The DescriptorHandle argument was an IRD.

[4] The DescriptorHandle argument was the same as the DescriptorHandle argument in the SQLGetDescField or SQLGetDescRec function that is running asynchronously.

[5] The DescriptorHandle argument was different than the DescriptorHandle argument in the SQLGetDescField or SQLGetDescRec function that is running asynchronously.

### SQLGetDescField and SQLGetDescRec (Prepared States)

<table>
<thead>
<tr>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Results</td>
<td>Results</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>

[1] The DescriptorHandle argument was an APD or ARD.

[2] The DescriptorHandle argument was an IPD.

[3] The DescriptorHandle argument was an IRD. Note that these functions always return SQL_NO_DATA in state S2 when DescriptorHandle was an IRD.

### SQLGetDiagField and SQLGetDiagRec

<table>
<thead>
<tr>
<th></th>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>--[1]</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

[1] This row shows transitions when HandleType was SQL_HANDLE_ENV, SQL_HANDLE_DBC, or SQL_HANDLE_DESC.

[2] This row shows transitions when HandleType was SQL_HANDLE_STMT.

[3] SQLGetDiagField always returns an error in this state when DiagIdentifier is SQL_DIAG_ROW_COUNT.

### SQLGetEnvAttr

<table>
<thead>
<tr>
<th></th>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

### SQLGetFunctions

<table>
<thead>
<tr>
<th></th>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
### SQLGetInfo

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

### SQLGetStmtAttr

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
</tbody>
</table>

[1] The statement attribute was not SQL_ATTR_ROW_NUMBER.

[2] The statement attribute was SQL_ATTR_ROW_NUMBER.

### SQLGetStmtAttr (Cursor States)

<table>
<thead>
<tr>
<th></th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opened</td>
<td>SQLFetch or SQLFetchScroll</td>
<td>SQLExtendedFetch</td>
</tr>
</tbody>
</table>

[1] The Attribute argument was not SQL_ATTR_ROW_NUMBER.

[2] The Attribute argument was SQL_ATTR_ROW_NUMBER.

### SQLMoreResults

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
</tbody>
</table>

[1] The function always returns SQL_NO_DATA in this state.

[2] The next result is a row count.

[3] The next result is a result set.

[4] The current result is the last result.

### SQLNativeSql

<table>
<thead>
<tr>
<th></th>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
SQLNumParams

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
<td>-- [s] S11 [x]</td>
<td>-- [s] S11 [x]</td>
<td>-- [s] S11 [x]</td>
<td>(HY010)</td>
<td>NS [c] (HY010) [o]</td>
</tr>
</tbody>
</table>

SQLNumResultCols

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
<td>-- [s] S11 [x]</td>
<td>-- [s] S11 [x]</td>
<td>-- [s] S11 [x]</td>
<td>(HY010)</td>
<td>NS [c] (HY010) [o]</td>
</tr>
</tbody>
</table>

SQLParamData

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>See next table</td>
<td>NS [c] (HY010) [o]</td>
<td></td>
</tr>
</tbody>
</table>

SQLParamData (Need Data States)

<table>
<thead>
<tr>
<th>S8</th>
<th>S9</th>
<th>S10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need Data</td>
<td>Must Put</td>
<td>Can Put</td>
</tr>
<tr>
<td>S1 [e] and [1] S2 [e], [nr], and [2] S3 [e], [r], and [2] S5 [e] and [4] S6 [e] and [5] S7 [e] and [3] S9 [d] S11 [x]</td>
<td>HY010</td>
<td>S1 [e] and [1] S2 [e], [nr], and [2] S3 [e], [r], and [2] S4 [s], [nr], and ([1] or [2]) S5 [s], [r], and ([1] or [2]) S5 ([s] or [e]) and [4] S6 ([s] or [e]) and [5] S7 ([s] or [e]) and [3] S9 [d] S11 [x]</td>
</tr>
</tbody>
</table>

[1] SQLExecDirect returned SQL_NEED_DATA.

[2] SQLExecute returned SQL_NEED_DATA.

[3] SQLSetPos had been called from state S7 and returned SQL_NEED_DATA.

[4] SQLBulkOperations had been called from state S5 and returned SQL_NEED_DATA.

[5] SQLSetPos or SQLBulkOperations had been called from state S6 and returned SQL_NEED_DATA.

SQLPrepare

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
</tbody>
</table>

[1] The preparation fails for a reason other than validating the statement (the SQLSTATE was HY009 [Invalid argument value] or HY090 [Invalid string or buffer length]).

[2] The preparation fails while validating the statement (the SQLSTATE was not HY009 [Invalid argument value] or HY090 [Invalid string or buffer length]).

[3] The current result is the last or only result, or there are no current results. For more information about multiple results, see Multiple Results.
The current result is not the last result.

### SQLPrepare (Cursor States)

<table>
<thead>
<tr>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opened</td>
<td>SQLFetch or SQLFetchScroll</td>
<td>SQLExtendedFetch</td>
</tr>
<tr>
<td>24000</td>
<td>(24000)</td>
<td>(24000)</td>
</tr>
</tbody>
</table>

### SQLPutData

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>See next table</td>
<td>NS [c] (HY010) [o]</td>
</tr>
</tbody>
</table>

### SQLPutData (Need Data States)

<table>
<thead>
<tr>
<th>S8</th>
<th>S9</th>
<th>S10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need Data</td>
<td>Must Put</td>
<td>Can Put</td>
</tr>
</tbody>
</table>

[1] SQLExecDirect returned SQL_NEED_DATA.
[2] SQLExecute returned SQL_NEED_DATA.
[3] SQLSetPos had been called from state S7 and returned SQL_NEED_DATA.
[4] SQLBulkOperations had been called from state S5 and returned SQL_NEED_DATA.
[5] SQLSetPos or SQLBulkOperations had been called from state S6 and returned SQL_NEED_DATA.
[6] One or more calls to SQLPutData for a single parameter returned SQL_SUCCESS, and then a call to SQLPutData was made for the same parameter with StrLen_or_Ind set to SQL_NULL_DATA.

### SQLRowCount

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
<td>(HY010)</td>
</tr>
</tbody>
</table>

### SQLSetConnectAttr

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>S2–S3</th>
<th>S4</th>
<th>S5–S7</th>
<th>S8–S10</th>
<th>S11–S12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Allocated</td>
<td>Prepared</td>
<td>Executed</td>
<td>Cursor</td>
<td>Need Data</td>
<td>Async</td>
</tr>
</tbody>
</table>

[1] This row shows transitions when Attribute was a connection attribute. For transitions when Attribute was a statement attribute, see the statement transition table for SQLSetStmtAttr.
[2] The Attribute argument was not SQL_ATTR_CURRENT_CATALOG.
[3] The Attribute argument was SQL_ATTR_CURRENT_CATALOG.
### SQLSetCursorName

<table>
<thead>
<tr>
<th></th>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td>--</td>
<td>--</td>
<td>(24000)</td>
<td>(24000)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td></td>
</tr>
</tbody>
</table>

### SQLSetDescField and SQLSetDescRec

<table>
<thead>
<tr>
<th></th>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)[1]</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[1] This row shows transitions where the `DescriptorHandle` argument is an ARD, APD, IPD, or (for `SQLSetDescField`) an IRD when the `FieldIdentifier` argument is `SQL_DESC_ARRAY_STATUS_PTR` or `SQL_DESC_ROWS_PROCESSED_PTR`. It is an error to call `SQLSetDescField` for an IRD when `FieldIdentifier` is any other value.

### SQLSetEnvAttr

<table>
<thead>
<tr>
<th></th>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
</table>

### SQLSetPos

<table>
<thead>
<tr>
<th></th>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td>(HY010)</td>
<td>(HY010)</td>
<td>(24000)</td>
<td>See next table</td>
<td>(HY010)</td>
<td>NS [c] (HY010) [s]</td>
<td></td>
</tr>
</tbody>
</table>

### SQLSetPos (Cursor States)

<table>
<thead>
<tr>
<th></th>
<th>S5 Opened</th>
<th>S6 <code>SQLFetch</code> or <code>SQLFetchScroll</code></th>
<th>S7 <code>SQLExtendedFetch</code></th>
</tr>
</thead>
</table>

### SQLSetStmtAttr

<table>
<thead>
<tr>
<th></th>
<th>S0 Unallocated</th>
<th>S1 Allocated</th>
<th>S2–S3 Prepared</th>
<th>S4 Executed</th>
<th>S5–S7 Cursor</th>
<th>S8–S10 Need Data</th>
<th>S11–S12 Async</th>
</tr>
</thead>
</table>

[1] The `Attribute` argument was not `SQL_ATTR_CONCURRENCY`, `SQL_ATTR_CURSOR_TYPE`, `SQL_ATTR_SIMULATE_CURSOR`, `SQL_ATTR_USE_BOOKMARKS`, `SQL_ATTR_CURSOR_SCROLLABLE`, or `SQL_ATTR_CURSOR_SENSITIVITY`.

[2] The `Attribute` argument was `SQL_ATTR_CONCURRENCY`, `SQL_ATTR_CURSOR_TYPE`, `SQL_ATTR_SIMULATE_CURSOR`, `SQL_ATTR_USE_BOOKMARKS`, `SQL_ATTR_CURSOR_SCROLLABLE`, or `SQL_ATTR_CURSOR_SENSITIVITY`. 
Descriptor Transitions

ODBC descriptors have the following three states.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>Unallocated descriptor</td>
</tr>
<tr>
<td>D1i</td>
<td>Implicitly allocated descriptor</td>
</tr>
<tr>
<td>D1e</td>
<td>Explicitly allocated descriptor</td>
</tr>
</tbody>
</table>

The following tables show how each ODBC function affects the descriptor state.

**SQLAllocHandle**

<table>
<thead>
<tr>
<th>D0 Unallocated</th>
<th>D1i Implicit</th>
<th>D1e Explicit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[1] This row shows transitions when HandleType was SQL_HANDLE_STMT.
[2] This row shows transitions when HandleType was SQL_HANDLE_DESC.

**SQLCopyDesc**

<table>
<thead>
<tr>
<th>D0 Unallocated</th>
<th>D1i Implicit</th>
<th>D1e Explicit</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SQLFreeHandle**

<table>
<thead>
<tr>
<th>D0 Unallocated</th>
<th>D1i Implicit</th>
<th>D1e Explicit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D0</td>
<td></td>
</tr>
<tr>
<td>(IH)[2]</td>
<td>(HY017)</td>
<td>D0</td>
</tr>
</tbody>
</table>

[1] This row shows transitions when HandleType was SQL_HANDLE_STMT.
[2] This row shows transitions when HandleType was SQL_HANDLE_DESC.

**SQLGetDescField and SQLGetDescRec**

<table>
<thead>
<tr>
<th>D0 Unallocated</th>
<th>D1i Implicit</th>
<th>D1e Explicit</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IH)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SQLSetDescField and SQLSetDescRec

<table>
<thead>
<tr>
<th></th>
<th>D0</th>
<th>D1i</th>
<th>D1e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Unallocated</td>
<td>Unallocated</td>
<td>Unallocated</td>
</tr>
</tbody>
</table>

([IH][1] -- --)

[1] This row shows transitions when DescriptorHandle was the handle of an ARD, APD, or IPD, or (for SQLSetDescField) when DescriptorHandle was the handle of an IRD and FieldIdentifier was SQL_DESC_ARRAY_STATUS_PTR or SQL_DESC_ROWS_PROCESSED_PTR.

All Other ODBC Functions

<table>
<thead>
<tr>
<th></th>
<th>D0</th>
<th>D1i</th>
<th>D1e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unallocated</td>
<td>Unallocated</td>
<td>Unallocated</td>
<td>Unallocated</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

Appendix C: SQL Grammar

This appendix contains the following topics.

- SQL Minimum Grammar
- ODBC Escape Sequences
- Literals in ODBC
- Reserved Keywords

© 2016 Microsoft

SQL Minimum Grammar

This section describes the minimum SQL syntax that an ODBC driver must support. The syntax described in this section is a subset of the Entry level syntax of SQL-92.

An application can use any of the syntax in this section and be assured that any ODBC-compliant driver will support that syntax. To determine whether additional features of SQL-92 not in this section are supported, the application should call SQLGetInfo with the SQL_SQL_CONFORMANCE information type. Even if the driver does not conform to any SQL-92 conformance level, an application can still use the syntax described in this section. If a driver conforms to an SQL-92 level, on the other hand, it supports all syntax included in that level. This includes the syntax in this section because the minimum grammar described here is a pure subset of the lowest SQL-92 conformance level. Once the application knows the SQL-92 level supported, it can determine whether a higher-level feature is supported (if any) by calling SQLGetInfo with the individual information type corresponding to that feature.

Drivers that work only with read-only data sources might not support those parts of the grammar included in this section that deal with changing data. An application can determine if a data source is read-only by calling SQLGetInfo with the SQL_DATA_SOURCE_READ_ONLY information type.

Statement

create-table-statement ::= CREATE TABLE base-table-name (column-identifier data-type [, column-identifier data-type]...)

Important

As a data-type in a create-table-statement, applications must use a data type from the TYPE_NAME column of the result set returned by SQLGetTypeInfo.
DELETE FROM table-name [WHERE search-condition]

drop-table-statement ::= DROP TABLE base-table-name

insert-statement ::= INSERT INTO table-name [( { column-identifier [, column-identifier]...} )] VALUES (insert-value[, insert-value]... )

select-statement ::= SELECT [ALL | DISTINCT] select-list
   FROM table-reference-list
   [WHERE search-condition]
   [order-by-clause]

statement ::= create-table-statement
   | delete-statement-searched
   | drop-table-statement
   | insert-statement
   | select-statement
   | update-statement-searched

update-statement-searched
UPDATE table-name
SET column-identifier = { expression | NULL }
   [, column-identifier = { expression | NULL} ]...
   [WHERE search-condition]

This section contains the following topics.

- Elements Used in SQL Statements
- Data Type Support
- Parameter Data Types
- Parameter Markers

Elements Used in SQL Statements

The following elements are used in the SQL statements listed previously.

Element

base-table-identifier ::= user-defined-name

base-table-name ::= base-table-identifier

boolean-factor ::= [NOT] boolean-primary

boolean-primary ::= comparison-predicate | ( search-condition )

boolean-term ::= boolean-factor [ AND boolean-term]

character-string-literal ::= "(character)"... (character is any character in the character set of the driver/data source. To include a single literal quote character (" ) in a character-string-literal, use two literal quote characters (""").)

column-identifier ::= user-defined-name

column-name ::= [table-name.]column-identifier

comparison-operator ::= < | > | <= | >= | = | <>

comparison-predicate ::= expression comparison-operator expression
data-type ::= character-string-type (character-string-type is any data type for which the "DATA_TYPE" column in the result set returned by SQLGetTypeInfo is either SQL_CHAR or SQL_VARCHAR.)
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
dynamic-parameter ::= ?
expression ::= term | expression \{+\,-\} term
factor ::= \+[\+\-]primary
insert-value ::= 
  dynamic-parameter
  | literal
  | NULL
  | USER
letter ::= lower-case-letter | upper-case-letter
literal ::= character-string-literal
lower-case-letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
order-by-clause ::= ORDER BY sort-specification [, sort-specification]...
primary ::= column-name
  | dynamic-parameter
  | literal
  | ( expression )
search-condition ::= boolean-term [OR search-condition]
select-list ::= * | select-sublist [, select-sublist]... (select-list cannot contain parameters.)
select-sublist ::= expression
sort-specification ::= \{unsigned-integer | column-name\} [ASC | DESC]
table-identifier ::= user-defined-name
table-name ::= table-identifier
table-reference ::= table-name
table-reference-list ::= table-reference [,table-reference]...
term ::= factor | term \{\*|/\} factor
unsigned-integer ::= \{digit\}
user-defined-name ::= letter[ digit | letter | _ ]...

Data Type Support

ODBC drivers must support at least one of SQL_CHAR and SQL_VARCHAR. Support for other data types is determined by the driver's or data source's SQL-92 conformance level. An application should call SQLGetTypeInfo to determine the data types supported by the driver.

For more information on data types, see Appendix D: Data Types.

Parameter Data Types

© 2016 Microsoft
Even though each parameter specified with `SQLBindParameter` is defined using an SQL data type, the parameters in an SQL statement have no intrinsic data type. Therefore, parameter markers can be included in an SQL statement only if their data types can be inferred from another operand in the statement. For example, in an arithmetic expression such as `? + COLUMN1`, the data type of the parameter can be inferred from the data type of the named column represented by COLUMN1. An application cannot use a parameter marker if the data type cannot be determined.

The following table describes how a data type is determined for several types of parameters, in accordance with SQL-92. For a more comprehensive specification on inferring the parameter type when other SQL clauses are used, see the SQL-92 specification.

<table>
<thead>
<tr>
<th>Location of parameter</th>
<th>Assumed data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>One operand of a binary arithmetic or comparison operator</td>
<td>Same as the other operand</td>
</tr>
<tr>
<td>The first operand in a <code>BETWEEN</code> clause</td>
<td>Same as the second operand</td>
</tr>
<tr>
<td>The second or third operand in a <code>BETWEEN</code> clause</td>
<td>Same as the first operand</td>
</tr>
<tr>
<td>An expression used with <code>IN</code></td>
<td>Same as the first value or the result column of the subquery</td>
</tr>
<tr>
<td>A value used with <code>IN</code></td>
<td>Same as the expression or the first value if there is a parameter marker in the expression</td>
</tr>
<tr>
<td>A pattern value used with <code>LIKE</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td>An update value used with <code>UPDATE</code></td>
<td>Same as the update column</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**Parameter Markers**

In accordance with the SQL-92 specification, an application cannot place parameter markers in the following locations. For a more comprehensive list, see the SQL-92 specification.

- In a `SELECT` list
- As both expressions in a `comparison-predicate`
- As both operands of a binary operator
- As both the first and second operands of a `BETWEEN` operation
- As both the first and third operands of a `BETWEEN` operation
- As both the expression and the first value of an `IN` operation
- As the operand of a unary `+` or `-` operation
- As the argument of a `set-function-reference`

For more information about parameter markers, see the SQL-92 specification. For more information about parameters, see [Statement Parameters](#).

© 2016 Microsoft

**ODBC Escape Sequences**

The following escape sequences are used in ODBC. The grammar in this section uses some elements defined in [Elements Used in SQL Statements](#).

This section contains the following topics.

- Date, Time, and Timestamp Escape Sequences
- GUID Escape Sequences
- Interval Escape Sequences
- `LIKE` Escape Sequence
- Outer Join Escape Sequence
- Procedure Call Escape Sequence
- Scalar Function Escape Sequence
Date, Time, and Timestamp Escape Sequences

ODBC defines escape sequences for date, time, and timestamp literals. The syntax of these escape sequences is as follows:

```
{d 'value'}
{t 'value'}
{ts 'value'}
```

In BNF notation, the syntax is as follows:

```
ODBC-date-time-escape ::= 
   ODBC-date-escape 
   | ODBC-time-escape
   | ODBC-timestamp-escape

ODBC-date-escape ::= 
   ODBC-esc-initiator d 'date-value' ODBC-esc-terminator

ODBC-time-escape ::= 
   ODBC-esc-initiator t 'time-value' ODBC-esc-terminator

ODBC-timestamp-escape ::= 
   ODBC-esc-initiator ts 'timestamp-value' ODBC-esc-terminator

ODBC-esc-initiator ::= \{

ODBC-esc-terminator ::= \}

date-value ::= 
   years-value date-separator months-value date-separator days-value

time-value ::= 
   hours-value time-separator minutes-value time-separator seconds-value

timestamp-value ::= 
   date-value timestamp-separator time-value

timestamp-separator ::= (The blank character)

years-value ::= digit digit digit digit

months-value ::= digit digit

days-value ::= digit digit

hours-value ::= digit digit

minutes-value ::= digit digit

seconds-value ::= digit digit [.digit...]
```

Remarks

The date, time, and timestamp literal escape sequences are supported if the date, time, and timestamp data types are supported by the data source. An application should call `SQLGetTypeInfo` to determine whether these data types are supported.

GUID Escape Sequences

ODBC uses escape sequences for GUID literals. The syntax of this escape sequence is as follows:

```
{guid 'nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn'}
```

Remarks

In BNF notation, the syntax is as follows:

```
ODBC-guid-escape ::= ODBC-esc-initiator guid 'guid-value' ODBC-esc-terminator

ODBC-esc-initiator ::= \{

ODBC-esc-terminator ::= \}

guid-value ::= clock-low-value guid-separator clock-middle-value guid-separator clock-high-value guid-separator clock-seq-value guid-separator node-value

guid-separator ::= -

clock-low-value ::= hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit

clock-middle-value ::= hex_digit hex_digit hex_digit hex_digit hex_digit

clock-high-value ::= hex_digit hex_digit hex_digit hex_digit

clock-seq-value ::= hex_digit hex_digit hex_digit hex_digit

clock-node-value ::= hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit

hex_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F
```

The GUID literal escape sequence is supported if the GUID data type is supported by the data source. An application should call `SQLGetTypeInfo` to determine whether this data type is supported.
Interval Escape Sequences

ODBC uses escape sequences for interval literals. The syntax of this escape sequence is as follows:

\{(interval-literal)\}

For the BNF syntax of `interval-literal`, see the Interval Literal Syntax section later in this appendix.

The interval literal escape sequence is supported if the interval data types are supported by the data source. An application should call `SQLGetTypeInfo` to determine whether these data types are supported.

LIKE Escape Sequence

ODBC uses escape sequences for the LIKE clause. The syntax of this escape sequence is as follows:

\{('escape-character')\}

Remarks

In BNF notation, the syntax is as follows:

```
ODBC-like-escape ::= 
    ODBC-esc-initiator escape 'escape-character' ODBC-esc-terminator
escape-character ::= character
ODBC-esc-initiator ::= { 
ODBC-esc-terminator ::= }
```

To determine if the driver supports the LIKE escape sequence, an application can call `SQLGetInfo` with the SQL_LIKE_ESCAPE_CLAUSE information type.

Outer Join Escape Sequence

ODBC uses escape sequences for outer joins. The syntax of this escape sequence is as follows:

\{oj outer-join\}

Remarks

In BNF notation, the syntax is as follows:

```
ODBC-outer-join-escape ::= 
    ODBC-esc-initiator oj outer-join ODBC-esc-terminator
outer-join ::= table-name [correlation-name] {LEFT | RIGHT | FULL}
    OUTER JOIN (table-name [correlation-name] | outer-join) ON
    search-condition
    correlation-name ::= user-defined-name
```
To determine which parts of this statement are supported, an application calls `SQLGetInfo` with the SQL_OJ_CAPABILITIES information type. For outer joins, search-condition must contain only the join condition between the specified table-names.

© 2016 Microsoft

Procedure Call Escape Sequence

ODBC uses escape sequences for procedure calls. The syntax of this escape sequence is as follows:
```
{{[?=}call procedure-name[([parameter][,[parameter]]...)]}}
```
In BNF notation, the syntax is as follows:
```
ODBC-procedure-escape ::= |
| ODBC-esc-initiator [?=] call procedure ODBC-esc-terminator
procedure ::= procedure-name | procedure-name (procedure-parameter-list)
procedure-identifier ::= user-defined-name
procedure-name ::= procedure-identifier
| owner-name.procedure-identifier
| catalog-name catalog-separator procedure-identifier
| catalog-name catalog-separator [owner-name].procedure-identifier
```
(The third syntax is valid only if the data source does not support owners.)
```
owner-name ::= user-defined-name
catalog-name ::= user-defined-name
catalog-separator ::= (implementation-defined)
```
(The catalog separator is returned through `SQLGetInfo` with the SQL_CATALOG_NAME_SEPARATOR information option.)
```
procedure-parameter-list ::= procedure-parameter
procedure-parameter ::= dynamic-parameter | literal | empty-string
empty-string ::= ODBC-esc-initiator ::={
ODBC-esc-terminator ::= }
(If a procedure parameter is an empty string, the procedure uses the default value for that parameter.)
```
To determine whether the data source supports procedures and the driver supports the ODBC procedure invocation syntax, an application can call `SQLGetInfo` with the SQL_PROCEDURES information type.

© 2016 Microsoft

Scalar Function Escape Sequence

ODBC uses escape sequences for scalar functions. The syntax of this escape sequence is as follows:
```
(fn scalar-function)
```
Remarks

In BNF notation, the syntax is as follows:
```
ODBC-scalar-function-escape ::= 
| ODBC-esc-initiator fn scalar-function ODBC-esc-terminator
```

scalar-function ::= function-name (argument-list)
(The definitions for the nonterminals function-name and function-name (argument-list) are derived from the list of scalar functions in Appendix E: Scalar Functions.)

ODBC-esc-initiator ::= {
    ODBC-esc-terminator ::= }

To determine whether the data source supports procedures and the driver supports the ODBC procedure invocation syntax, an application can call SQLGetInfo. For more information, see Appendix E: Scalar Functions.

© 2016 Microsoft

Literals in ODBC

The syntax in the following sections is used for interval and numeric literals in ODBC. This syntax is provided here as an aid to driver writers when conversions are performed from a character string type to a numeric or interval type, or from a numeric or interval type to a character string type. For more information, see Interval Literals and Numeric Literals in Appendix D: Data Types.

This appendix contains the following topics.

- Interval Literal Syntax
- Numeric Literal Syntax

© 2016 Microsoft

Interval Literal Syntax

The following syntax is used for interval literals in ODBC.

interval-literal ::= INTERVAL [+|-] interval-string interval-qualifier
interval-string ::= quote { year-month-literal | day-time-literal } quote
year-month-literal ::= years-value | [years-value -] months-value
day-time-literal ::= day-time-interval | time-interval
day-time-interval ::= days-value [hours-value [::minutes-value<::seconds-value]]
time-interval ::= hours-value [::minutes-value ::seconds-value ]
    | minutes-value ::seconds-value
    | seconds-value
years-value ::= datetime-value
months-value ::= datetime-value
days-value ::= datetime-value
hours-value ::= datetime-value
minutes-value ::= datetime-value
seconds-value ::= seconds-integer-value [::seconds-fraction]
seconds-integer-value ::= unsigned-integer
seconds-fraction ::= unsigned-integer
datetime-value ::= unsigned-integer
interval-qualifier ::= start-field TO end-field | single-datetime-field
start-field ::= non-second-datetime-field [[interval-leading-field-precision ]]
end-field ::= non-second-datetime-field | SECOND[(interval-fractional-seconds-precision)]
single-datetime-field ::= non-second-datetime-field [[interval-leading-field-precision]] | SECOND[(interval-leading-field-precision [, (interval-fractional-seconds-precision)]]
Numeric Literal Syntax

The following syntax is used for numeric literals in ODBC:

- `numeric-literal ::= signed-numeric-literal | unsigned-numeric-literal`
- `signed-numeric-literal ::= [sign] unsigned-numeric-literal`
- `unsigned-numeric-literal ::= exact-numeric-literal | approximate-numeric-literal`
- `exact-numeric-literal ::= unsigned-integer [period unsigned-integer] | period unsigned-integer`
- `sign ::= plus-sign | minus-sign`
- `approximate-numeric-literal ::= mantissa E exponent`
- `mantissa ::= exact-numeric-literal`
- `exponent ::= signed-integer`
- `unsigned-integer ::= [sign] unsigned-integer`
- `plus-sign ::= +`
- `minus-sign ::= -`
- `digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0`
- `period ::= .`

Reserved Keywords

The following words are reserved for use in ODBC function calls. These words do not constrain the minimum SQL grammar; however, to ensure compatibility with drivers that support the core SQL grammar, applications should avoid using any of these keywords. The `#define` value SQL_ODBC_KEYWORDS contains a comma-separated list of these keywords.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSOLUTE</td>
<td></td>
</tr>
<tr>
<td>ACTION</td>
<td>IS</td>
</tr>
<tr>
<td>ADA</td>
<td>JOIN</td>
</tr>
<tr>
<td>ADD</td>
<td>KEY</td>
</tr>
<tr>
<td>ALL</td>
<td>LANGUAGE</td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>LAST</td>
</tr>
<tr>
<td>ALTER</td>
<td>LEADING</td>
</tr>
<tr>
<td>AND</td>
<td>LEFT</td>
</tr>
<tr>
<td>ANY</td>
<td>LEVEL</td>
</tr>
<tr>
<td>ARE</td>
<td>LIKE</td>
</tr>
<tr>
<td>AS</td>
<td>LOCAL</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>ASC</td>
<td>LOWER</td>
</tr>
<tr>
<td>ASSERTION</td>
<td>MATCH</td>
</tr>
<tr>
<td>AT</td>
<td>MAX</td>
</tr>
<tr>
<td>AUTHORIZATION</td>
<td>MIN</td>
</tr>
<tr>
<td>AVG</td>
<td>MINUTE</td>
</tr>
<tr>
<td>BEGIN</td>
<td>MODULE</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>MONTH</td>
</tr>
<tr>
<td>BIT</td>
<td>NAMES</td>
</tr>
<tr>
<td>BIT_LENGTH</td>
<td>NATIONAL</td>
</tr>
<tr>
<td>BOTH</td>
<td>NATURAL</td>
</tr>
<tr>
<td>BY</td>
<td>NCHAR</td>
</tr>
<tr>
<td>CASCADE</td>
<td>NEXT</td>
</tr>
<tr>
<td>CASCADED</td>
<td>NO</td>
</tr>
<tr>
<td>CASE</td>
<td>NONE</td>
</tr>
<tr>
<td>CAST</td>
<td>NOT</td>
</tr>
<tr>
<td>CATALOG</td>
<td>NULL</td>
</tr>
<tr>
<td>CHAR</td>
<td>NULLIF</td>
</tr>
<tr>
<td>CHAR_LENGTH</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>OCTET_LENGTH</td>
</tr>
<tr>
<td>CHARACTER_LENGTH</td>
<td>OF</td>
</tr>
<tr>
<td>CHECK</td>
<td>ON</td>
</tr>
<tr>
<td>CLOSE</td>
<td>ONLY</td>
</tr>
<tr>
<td>COALESCE</td>
<td>OPEN</td>
</tr>
<tr>
<td>COLLATE</td>
<td>OPTION</td>
</tr>
<tr>
<td>COLLATION</td>
<td>OR</td>
</tr>
<tr>
<td>COLUMN</td>
<td>ORDER</td>
</tr>
<tr>
<td>COMMIT</td>
<td>OUTER</td>
</tr>
<tr>
<td>CONNECT</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>CONNECTION</td>
<td>OVERLAPS</td>
</tr>
<tr>
<td>CONSTRAINT</td>
<td>PAD</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>PARTIAL</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>PASCAL</td>
</tr>
<tr>
<td>CONVERT</td>
<td>POSITION</td>
</tr>
<tr>
<td>CORRESPONDING</td>
<td>PRECISION</td>
</tr>
<tr>
<td>COUNT</td>
<td>PREPARE</td>
</tr>
<tr>
<td>CREATE</td>
<td>PRESERVE</td>
</tr>
<tr>
<td>CROSS</td>
<td>PRIMARY</td>
</tr>
<tr>
<td>CURRENT_DATE</td>
<td>PRIVILEGES</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>CURRENT_TIME</td>
<td>PROCEDURE</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>PUBLIC</td>
</tr>
<tr>
<td>CURRENT_USER</td>
<td>READ</td>
</tr>
<tr>
<td>CURSOR</td>
<td>REAL</td>
</tr>
<tr>
<td>DATE</td>
<td>REFERENCES</td>
</tr>
<tr>
<td>DAY</td>
<td>RELATIVE</td>
</tr>
<tr>
<td>DEALLOCATE</td>
<td>RESTRICT</td>
</tr>
<tr>
<td>DEC</td>
<td>REVOKE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>RIGHT</td>
</tr>
<tr>
<td>DECLARE</td>
<td>ROLLBACK</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>ROWS</td>
</tr>
<tr>
<td>DEFERRABLE</td>
<td>SCHEMA</td>
</tr>
<tr>
<td>DEFERRED</td>
<td>SCROLL</td>
</tr>
<tr>
<td>DELETE</td>
<td>SECOND</td>
</tr>
<tr>
<td>DESC</td>
<td>SECTION</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>SELECT</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
<td>SESSION</td>
</tr>
<tr>
<td>DIAGNOSTICS</td>
<td>SESSION_USER</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>SET</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>SIZE</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>SOME</td>
</tr>
<tr>
<td>DROP</td>
<td>SPACE</td>
</tr>
<tr>
<td>ELSE</td>
<td>SQL</td>
</tr>
<tr>
<td>END</td>
<td>SQLCA</td>
</tr>
<tr>
<td>END-EXEC</td>
<td>SQLCODE</td>
</tr>
<tr>
<td>ESCAPE</td>
<td>SQLERROR</td>
</tr>
<tr>
<td>EXCEPT</td>
<td>SQLSTATE</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>SQLWARNING</td>
</tr>
<tr>
<td>EXEC</td>
<td>SUBSTRING</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>SUM</td>
</tr>
<tr>
<td>EXISTS</td>
<td>SYSTEM_USER</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>TABLE</td>
</tr>
<tr>
<td>EXTRACT</td>
<td>TEMPORARY</td>
</tr>
<tr>
<td>FALSE</td>
<td>THEN</td>
</tr>
<tr>
<td>FETCH</td>
<td>TIME</td>
</tr>
</tbody>
</table>
Appendix D: Data Types

ODBC defines two sets of data types: SQL data types and C data types. SQL data types indicate the data type of data stored at the data source. C data types indicate the data type of data stored in application buffers.
SQL data types are defined by each DBMS in accordance with the SQL-92 standard. For each SQL data type specified in the SQL-92 standard, ODBC defines a type identifier, which is a `#define` value that is passed as an argument in ODBC functions or returned in the metadata of a result set. The only SQL-92 data types not supported by ODBC are BIT (the ODBC SQL_BIT type has different characteristics), BIT_VARYING, TIME_WITH_TIMEZONE, TIMESTAMP_WITH_TIMEZONE, and NATIONAL_CHARACTER. Drivers are responsible for mapping data source-specific SQL data types to ODBC SQL data type identifiers and driver-specific SQL data type identifiers. The SQL data type is specified in the SQL_DESC_CONCISE_TYPE field of an implementation descriptor.

ODBC defines the C data types and their corresponding ODBC type identifiers. An application specifies the C data type of the buffer that will receive result set data by passing the appropriate C type identifier in the `TargetType` argument in a call to `SQLBindCol` or `SQLGetData`. It specifies the C type of the buffer containing a statement parameter by passing the appropriate C type identifier in the `ValueType` argument in a call to `SQLBindParameter`. The C data type is specified in the SQL_DESC_CONCISE_TYPE field of an application descriptor.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are no driver-specific C data types.</td>
</tr>
</tbody>
</table>

Each SQL data type corresponds to an ODBC C data type. Before returning data from the data source, the driver converts it to the specified C data type. Before sending data to the data source, the driver converts it from the specified C data type.

This appendix contains the following topics.

- Using Data Type Identifiers
- SQL Data Types
- C Data Types
- Data Type Identifiers and Descriptors
- Pseudo-Type Identifiers
- Transferring Data in Its Binary Form
- Guidelines for Interval and Numeric Data Types
- Constraints of the Gregorian Calendar
- Column Size, Decimal Digits, Transfer Octet Length, and Display Size
- Converting Data from SQL to C Data Types
- Converting Data from C to SQL Data Types

For an explanation of ODBC data types, see Data Types in ODBC. For information about driver-specific SQL data types, see the driver’s documentation.

© 2016 Microsoft

Using Data Type Identifiers

Applications use data type identifiers in two ways: to describe their buffers to the driver, and to retrieve metadata about the result set from the driver so that they can determine what type of C buffers to use to store the data. Applications call the following functions to perform these tasks:

- `SQLBindParameter`, `SQLBindCol`, and `SQLGetData` — to describe the C data type of application buffers.
- `SQLBindParameter` — to describe the SQL data type of dynamic parameters.
- `SQLColAttribute` and `SQLDescribeCol` — to retrieve the SQL data types of result set columns.
- `SQLDescribeParameter` — to retrieve the SQL data types of parameters.
- `SQLColumns`, `SQLProcedureColumns`, and `SQLSpecialColumns` — to retrieve the SQL data types of various schema information
- `SQLGetTypeInfo` — to retrieve a list of supported data types

Data type identifiers are stored in the SQL_DESC_CONCISE_TYPE field of a descriptor. The descriptor functions `SQLSetDescField` and `SQLSetDescRec` can be used with the appropriate types to perform the tasks listed in the previous list. For more information, see `SQLSetDescField`.

© 2016 Microsoft

SQL Data Types

Each DBMS defines its own SQL types. Each ODBC driver exposes only those SQL data types that the associated DBMS defines. Information about how a driver maps DBMS SQL types to the ODBC-defined SQL type identifiers and how a driver maps DBMS SQL types to its own driver-specific SQL type identifiers is returned through a call to `SQLGetTypeInfo`. A driver also returns the SQL data types when describing the data types of columns and parameters through calls to `SQLColAttribute`, `SQLColumns`, `SQLDescribeCol`, `SQLDescribeParam`, `SQLProcedureColumns`, and `SQLSpecialColumns`. 
The SQL data types are contained in the SQL_DESC_CONCISE_TYPE, SQL_DESC_TYPE, and SQL_DESC_DATETIME_INTERVAL_CODE fields of the implementation descriptors. Characteristics of the SQL data types are contained in the SQL_DESC_PRECISION, SQL_DESC_SCALE, SQL_DESC_LENGTH, and SQL_DESC_OCTET_LENGTH fields of the implementation descriptors. For more information, see Data Type Identifiers and Descriptors later in this appendix.

A given driver and data source do not necessarily support all the SQL data types that are defined in this appendix. A driver’s support for SQL data types depends on the level of SQL-92 that the driver complies with. To determine the level of SQL-92 grammar supported by the driver, an application calls SQLGetInfo with the SQL_SQL_CONFORMANCE information type. Additionally, a given driver and data source may support additional, driver-specific SQL data types. To determine which data types a driver supports, an application calls SQLGetTypeInfo. For information about driver-specific SQL data types, see the driver’s documentation. For information about the data types in a specific data source, see the documentation for that data source.

### Important

The tables throughout this appendix are only guidelines and show typically used names, ranges, and limits of SQL data types. A given data source might support only some of the listed data types, and the characteristics of the supported data types can differ from those listed.

The following table lists valid SQL type identifiers for all SQL data types. The table also lists the name and description of the corresponding data type from SQL-92 (if one exists).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>CHAR(n)</td>
<td>Character string of fixed string length n.</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>VARCHAR(n)</td>
<td>Variable-length character string with a maximum string length n.</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>LONG VARCHAR</td>
<td>Variable length character data. Maximum length is data source–dependent.[9]</td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>WCHAR(n)</td>
<td>Unicode character string of fixed string length n.</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>VARWCHAR(n)</td>
<td>Unicode variable-length character string with a maximum string length n.</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td>LONGWVARCHAR</td>
<td>Unicode variable-length character data. Maximum length is data source–dependent.</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>DECIMAL(p,s)</td>
<td>Signed, exact, numeric value with a precision of at least p and scale s. (The maximum precision is driver-defined.) (1 &lt;= p &lt;= 15; s &lt;= p).[4]</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>NUMERIC(p,s)</td>
<td>Signed, exact, numeric value with a precision p and scale s (1 &lt;= p &lt;= 15; s &lt;= p).[4]</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>SMALLINT</td>
<td>Exact numeric value with precision 5 and scale 0 (signed: −32,768 &lt;= n &lt;= 32,767; unsigned: 0 &lt;= n &lt;= 65,535).[3].</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>INTEGER</td>
<td>Exact numeric value with precision 10 and scale 0 (signed: −2[31] &lt;= n &lt;= 2[31] − 1; unsigned: 0 &lt;= n &lt;= 2[32] − 1).[3].</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>REAL</td>
<td>Signed, approximate, numeric value with a binary precision 24 (zero or absolute value 10[−38] to 10[38])..</td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td>FLOAT(p)</td>
<td>Signed, approximate, numeric value with a binary precision of at least p. (The maximum precision is driver-defined.).[5]</td>
</tr>
<tr>
<td>SQLDOUBLE</td>
<td>DOUBLE PRECISION</td>
<td>Signed, approximate, numeric value with a binary precision 53 (zero or absolute value 10[−308] to 10[308]).</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>BIT</td>
<td>Single bit binary data.[8]</td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td>TINYINT</td>
<td>Exact numeric value with precision 3 and scale 0 (signed: −128 &lt;= n &lt;= 127; unsigned: 0 &lt;= n &lt;= 255).[3].</td>
</tr>
<tr>
<td>SQL_BIGINT</td>
<td>BIGINT</td>
<td>Exact numeric value with precision 19 (if signed) or 20 (if unsigned) and scale 0 (signed: −2[63] &lt;= n &lt;= 2[63] − 1; unsigned: 0 &lt;= n &lt;= 2[64] − 1).[3].</td>
</tr>
<tr>
<td>SQL_BINARY</td>
<td>BINARY(n)</td>
<td>Binary data of fixed length n.[9]</td>
</tr>
<tr>
<td>SQL_VARBINARY</td>
<td>VARBINARY(n)</td>
<td>Variable length binary data of maximum length n. The maximum is set by the user.[9]</td>
</tr>
<tr>
<td>SQL_LONGVARBINARY</td>
<td>LONG VARBINARY</td>
<td>Variable length binary data. Maximum length is data source–dependent.[9]</td>
</tr>
<tr>
<td>SQL_DATE[6]</td>
<td>DATE</td>
<td>Year, month, and day fields, conforming to the rules of the Gregorian calendar. (See Constraints of the Gregorian Calendar, later in this appendix.)</td>
</tr>
<tr>
<td>SQL_TYPE_TIME[6]</td>
<td>TIME(p)</td>
<td>Hour, minute, and second fields, with valid values for hours of 00 to 23, valid values for minutes of 00 to 59, and valid values for seconds of 00 to 61. Precision p indicates the seconds precision.</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP[6]</td>
<td>TIMESTAMP(p)</td>
<td>Year, month, day, hour, minute, and second fields, with valid values as defined for the DATE and TIME data types.</td>
</tr>
<tr>
<td>SQL_TYPE_UCTDATETIME</td>
<td>UTCDATETIME</td>
<td>Year, month, day, hour, minute, second, utchour, and utcminute fields. The utchour and utcminute fields have 1/10 microsecond precision.</td>
</tr>
<tr>
<td>SQL_TYPE_UTCTIME</td>
<td>UTCTIME</td>
<td>Hour, minute, second, utchour, and utcminute fields. The utchour and utcminute fields have 1/10 microsecond precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_MONTH[7]</td>
<td>INTERVAL MONTH(p)</td>
<td>Number of months between two dates; p is the interval leading precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_YEAR[7]</td>
<td>INTERVAL YEAR(p)</td>
<td>Number of years between two dates; p is the interval leading precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_YEAR_TO_MONTH[7]</td>
<td>INTERVAL YEAR(p) TO MONTH</td>
<td>Number of years and months between two dates; p is the interval leading precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY[7]</td>
<td>INTERVAL DAY(p)</td>
<td>Number of days between two dates; p is the interval leading precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_HOUR[7]</td>
<td>INTERVAL HOUR(p)</td>
<td>Number of hours between two date/times; p is the interval leading precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_MINUTE[7]</td>
<td>INTERVAL MINUTE(p)</td>
<td>Number of minutes between two date/times; p is the interval leading precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_SECOND[7]</td>
<td>INTERVAL SECOND(p,q)</td>
<td>Number of seconds between two date/times; p is the interval leading precision and q is the interval seconds precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_HOUR[7]</td>
<td>INTERVAL DAY(p) TO HOUR</td>
<td>Number of days/hours between two date/times; p is the interval leading precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_MINUTE[7]</td>
<td>INTERVAL DAY(p) TO MINUTE</td>
<td>Number of days/hours/minutes between two date/times; p is the interval leading precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_SECOND[7]</td>
<td>INTERVAL DAY(p) TO SECOND(q)</td>
<td>Number of days/hours/minutes/seconds between two date/times; p is the interval leading precision and q is the interval seconds precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_HOUR_TO_MINUTE[7]</td>
<td>INTERVAL HOUR(p) TO MINUTE</td>
<td>Number of hours/minutes between two date/times; p is the interval leading precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_HOUR_TO_SECOND[7]</td>
<td>INTERVAL HOUR(p) TO SECOND(q)</td>
<td>Number of hours/minutes/seconds between two date/times; p is the interval leading precision and q is the interval seconds precision.</td>
</tr>
<tr>
<td>SQL_INTERVAL_MINUTE_TO_SECOND[7]</td>
<td>INTERVAL MINUTE(p) TO SECOND(q)</td>
<td>Number of minutes/seconds between two date/times; p is the interval leading precision and q is the interval seconds precision.</td>
</tr>
<tr>
<td>SQL_GUID</td>
<td>GUID</td>
<td>Fixed length GUID.</td>
</tr>
</tbody>
</table>

[1] This is the value returned in the DATA_TYPE column by a call to SQLGetTypeInfo.

[2] This is the value returned in the NAME column and CREATE PARAMS column by a call to SQLGetTypeInfo. The NAME column returns the designation—for example, CHAR—whereas the CREATE PARAMS column returns a comma-separated list of creation parameters such as precision, scale, and length.

[3] An application uses SQLGetTypeInfo or SQLColAttribute to determine whether a particular data type or a particular column in a result set is unsigned.

[4] SQL_DECIMAL and SQL_NUMERIC data types differ only in their precision. The precision of a DECIMAL(p,s) is an implementation-defined decimal precision that is no less than p, whereas the precision of a NUMERIC(p,s) is exactly equal to p.

[5] Depending on the implementation, the precision of SQL_FLOAT can be either 24 or 53: if it is 24, the SQL_FLOAT data type is the same as SQL_REAL; if it is 53, the SQL_FLOAT data type is the same as SQL_DOUBLE.

[6] In ODBC 3.x, the SQL date, time, and timestamp data types are SQL_TYPE_DATE, SQL_TYPE_TIME, and SQL_TYPE_TIMESTAMP, respectively; in ODBC 2.x, the data types are SQL_DATE, SQL_TIME, and SQL_TIMESTAMP.

[7] For more information about the interval SQL data types, see the Interval Data Types section, later in this appendix.

[8] The SQL_BIT data type has different characteristics than the BIT type in SQL-92.
This data type has no corresponding data type in SQL-92.

This section provides the following example.

- **Example SQLGetTypeInfo Result Set**

An application calls `SQLGetTypeInfo` to determine which data types are supported by a data source and the characteristics of those data types. The following tables show a sample result set returned by `SQLGetTypeInfo` for a data source that supports SQL_CHAR, SQL_LONGVARCHAR, SQL_DECIMAL, SQL_REAL, SQL_DATETIME, SQL_INTERVAL_YEAR, and SQL_INTERVAL_DAY_TO_SECOND.

<table>
<thead>
<tr>
<th>TYPE_NAME</th>
<th>DATA_TYPE</th>
<th>COLUMN_SIZE</th>
<th>LITERAL_PREFIX</th>
<th>LITERAL_SUFFIX</th>
<th>CREATE_PARAMS</th>
<th>NULLABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;char&quot;</td>
<td>SQL_CHAR</td>
<td>255</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;length&quot;</td>
<td>SQL_TRUE</td>
</tr>
<tr>
<td>&quot;text&quot;</td>
<td>SQL_LONGVARCHAR</td>
<td>2147483647</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&lt;Null&gt;</td>
<td>SQL_TRUE</td>
</tr>
<tr>
<td>&quot;decimal&quot;</td>
<td>SQL_DECIMAL</td>
<td>28</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>&quot;precision, scale&quot;</td>
<td>SQL_TRUE</td>
</tr>
<tr>
<td>&quot;real&quot;</td>
<td>SQL_REAL</td>
<td>7</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>SQL_TRUE</td>
</tr>
<tr>
<td>&quot;datetime&quot;</td>
<td>SQL_TYPE_TIMESTAMP</td>
<td>23</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>SQL_TRUE</td>
</tr>
<tr>
<td>&quot;INTERVAL YEAR() TO YEAR&quot;</td>
<td>SQL_INTERVAL_YEAR</td>
<td>9</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>&quot;precision&quot;</td>
<td>SQL_TRUE</td>
</tr>
<tr>
<td>&quot;INTERVAL DAY() TO FRACTION(5)&quot;</td>
<td>SQL_INTERVAL_DAY_TO_SECOND</td>
<td>24</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>&quot;precision&quot;</td>
<td>SQL_TRUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA_TYPE</th>
<th>CASE_SENSITIVE</th>
<th>SEARCHABLE</th>
<th>UNSIGNED_ATTRIBUTE</th>
<th>FIXED_PRECSCALE</th>
<th>AUTO_UNIQUE_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>SQL_FALSE</td>
<td>SQL_SEARCHABLE</td>
<td>&lt;Null&gt;</td>
<td>SQL_FALSE</td>
<td>&lt;Null&gt;</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>SQL_FALSE</td>
<td>SQL_PRED_CHAR</td>
<td>&lt;Null&gt;</td>
<td>SQL_FALSE</td>
<td>&lt;Null&gt;</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>SQL_FALSE</td>
<td>SQL_PRED_BASIC</td>
<td>SQL_FALSE</td>
<td>SQL_FALSE</td>
<td>SQL_FALSE</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>SQL_FALSE</td>
<td>SQL_PRED_BASIC</td>
<td>&lt;Null&gt;</td>
<td>SQL_FALSE</td>
<td>&lt;Null&gt;</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>SQL_FALSE</td>
<td>SQL_SEARCHABLE</td>
<td>&lt;Null&gt;</td>
<td>SQL_FALSE</td>
<td>&lt;Null&gt;</td>
</tr>
<tr>
<td>SQL_INTERVAL_YEAR</td>
<td>SQL_FALSE</td>
<td>SQL_SEARCHABLE</td>
<td>&lt;Null&gt;</td>
<td>SQL_FALSE</td>
<td>&lt;Null&gt;</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_SECOND</td>
<td>SQL_FALSE</td>
<td>SQL_PRED_BASIC</td>
<td>&lt;Null&gt;</td>
<td>SQL_FALSE</td>
<td>&lt;Null&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA_TYPE</th>
<th>MINIMUM_SCALE</th>
<th>MAXIMUM_SCALE</th>
<th>SQL_DATA_TYPE</th>
<th>SQL_DATETIME_SUB</th>
<th>NUM_PREC_R</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>SQL_CHAR</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>SQL_LONGVARCHAR</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>0</td>
<td>28</td>
<td>SQL_DECIMAL</td>
<td>&lt;Null&gt;</td>
<td>10</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>&lt;Null&gt;</td>
<td>&lt;Null&gt;</td>
<td>SQL_REAL</td>
<td>&lt;Null&gt;</td>
<td>10</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>3</td>
<td>3</td>
<td>SQL_DATETIME</td>
<td>SQL_CODE_TIMESTAMP</td>
<td>&lt;Null&gt;</td>
</tr>
<tr>
<td>SQL_INTERVAL_YEAR</td>
<td>0</td>
<td>0</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_INTERVAL_YEAR</td>
<td>&lt;Null&gt;</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_SECOND</td>
<td>5</td>
<td>5</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_INTERVAL_DAY_TO_SECOND</td>
<td>&lt;Null&gt;</td>
</tr>
</tbody>
</table>

© 2016 Microsoft
C Data Types

ODBC C data types indicate the data type of C buffers used to store data in the application.

All drivers must support all C data types. This is required because all drivers must support all C types to which SQL types that they support can be converted, and all drivers support at least one character SQL type. Because the character SQL type can be converted to and from all C types, all drivers must support all C types.

The C data type is specified in the SQLBindCol and SQLGetData functions with the TargetType argument and in the SQLBindParameter function with the ValueType argument. It can also be specified by calling SQLSetDescField to set the SQL_DESC_CONCISE_TYPE field of an ARD or APD, or by calling SQLSetDescRec with the Type argument (and the SubType argument if needed) and the DescriptorHandle argument set to the handle of an ARD or APD.

The following table lists valid type identifiers for the C data types. The table also lists the ODBC C data type that corresponds to each identifier and the definition of this data type.

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>ODBC C typedef</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_CHAR</td>
<td>SQLCHAR *</td>
<td>unsigned char *</td>
</tr>
<tr>
<td>SQL_C_WCHAR</td>
<td>SQLWCHAR *</td>
<td>wchar_t *</td>
</tr>
<tr>
<td>SQL_C_SSHORT[j]</td>
<td>SQLSMALLINT</td>
<td>short int</td>
</tr>
<tr>
<td>SQL_C_USHORT[j]</td>
<td>SQLUSMALLINT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>SQL_C_SLONG[j]</td>
<td>SQLINTEGER</td>
<td>long int</td>
</tr>
<tr>
<td>SQL_C_U_LONG[j]</td>
<td>SQLINTENER</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>SQL_C_FLOAT</td>
<td>SQLREAL</td>
<td>float</td>
</tr>
<tr>
<td>SQL_C_DOUBLE</td>
<td>SQLDOUBLE, SQLFLOAT</td>
<td>double</td>
</tr>
<tr>
<td>SQL_C_BIT</td>
<td>SQLCHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>SQL_C_STINYINT[j]</td>
<td>SQLSCHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>SQL_C_UTINYINT[j]</td>
<td>SQLCHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>SQL_C_SBIGINT</td>
<td>SQLBIGINT</td>
<td>_int64[h]</td>
</tr>
<tr>
<td>SQL_C_UBIGINT</td>
<td>SQLUBIGINT</td>
<td>unsigned _int64[h]</td>
</tr>
<tr>
<td>SQL_C_BINARY</td>
<td>SQLCHAR *</td>
<td>unsigned char *</td>
</tr>
<tr>
<td>SQL_C_BOOKMARK[i]</td>
<td>BOOKMARK</td>
<td>unsigned long int [d]</td>
</tr>
<tr>
<td>SQL_C_VAR_BOOKMARK</td>
<td>SQLCHAR *</td>
<td>unsigned char *</td>
</tr>
</tbody>
</table>

**SQL_C_TYPE_DATE[c]**

```c
struct tagDATE_STRUCT {
    SQLSMALLINT year;
    SQLUSMALLINT month;
    SQLUSMALLINT day;
} DATE_STRUCT;[a]
```

**SQL_C_TYPE_TIME[c]**

```c
struct tagTIME_STRUCT {
    SQLUSMALLINT hour;
    SQLUSMALLINT minute;
    SQLUSMALLINT second;
} TIME_STRUCT;[a]
```

**SQL_C_TYPE_TIMESTAMP[c]**

```c
struct tagTIMESTAMP_STRUCT {
    SQLSMALLINT year;
    SQLUSMALLINT month;
    SQLUSMALLINT day;
```

[a]: These structures are defined in the ODBC header files.
All C interval data types

- SQL_INTERVAL_STRUCT

See the C Interval Structure section, later in this appendix.

---

[a] The values of the year, month, day, hour, minute, and second fields in the datetime C data types must conform to the constraints of the Gregorian calendar. (See Constraints of the Gregorian Calendar later in this appendix.)

[b] The value of the fraction field is the number of billonths of a second and ranges from 0 through 999,999,999 (1 less than 1 billion). For example, the value of the fraction field for a half-second is 500,000,000, for a thousandth of a second (one millisecond) is 1,000,000, for a milllionth of a second (one microsecond) is 1,000, and for a billonth of a second (one nanosecond) is 1.

[c] In ODBC 2.x, the C date, time, and timestamp data types are SQL_C_DATE, SQL_C_TIME, and SQL_C_TIMESTAMP.

[d] ODBC 3.x applications should use SQL_C_VARBOOKMARK, not SQL_C_BOOKMARK. When an ODBC 3.x application works with an ODBC 2.x driver, the ODBC 3.x Driver Manager will map SQL_C_VARBOOKMARK to SQL_C_BOOKMARK.

[e] A number is stored in the val field of the SQL_NUMERIC_STRUCT structure as a scaled integer, in little endian mode (the leftmost byte being the least-significant byte). For example, the number 10.001 base 10, with a scale of 4, is scaled to an integer of 100010. Because this is 186AA in hexadecimal format, the value in SQL_NUMERIC_STRUCT would be "AA 86 01 00 00 ... 00", with the number of bytes defined by the SQL_MAX_NUMERIC_LEN #define.

For more information about SQL_NUMERIC_STRUCT, see HOWTO: Retrieving Numeric Data with SQL_NUMERIC_STRUCT and INF: How to Use SQL_C_NUMERIC Data Type with Numeric Data.

[f] The precision and scale fields of the SQL_C_NUMERIC data type are used for input from an application and for output from the driver to the application. When the driver writes a numeric value into the SQL_NUMERIC_STRUCT, it will use its own driver-specific default as the value for the precision field, and it will use the value in the SQL_DESC_SCALE field of the application descriptor (which defaults to 0) for the scale field. An application can provide its own values for precision and scale by setting the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the application descriptor.

[g] The sign field is 1 if positive, 0 if negative.

[h] _int64 might not be supplied by some compilers.

[i] SQL_C_BOOKMARK has been deprecated in ODBC 3.x.

[j] SQL_C_SHORT, SQL_C_LONG, and SQL_C_TINYINT have been replaced in ODBC by signed and unsigned types: SQL_C_SSHORT and SQL_C_USHORT, SQL_C_SLONG and SQL_C_USLONG, and SQL_C_STINYINT and SQL_C_UTINYINT. An ODBC 3.x driver that should work with ODBC 2.x applications should support SQL_C_SHORT, SQL_C_LONG, and SQL_C_TINYINT, because when they are called, the Driver Manager passes them through to the driver.

[k] SQL_C_GUID can be converted only to SQL_CHAR or SQL_WCHAR.

This section contains the following topic.

- 64-Bit Integer Structures

See Also

- C Data Types in ODBC

© 2016 Microsoft

64–Bit Integer Structures
The C type for the SQL_C_SBIGINT and SQL_C_UBIGINT data type identifiers on Microsoft C compilers is _int64. When a compiler other than a Microsoft® C compiler is used, the C type might be different. If the compiler supports 64-bit integers natively, the driver or application should define ODBCINT64 to be the native 64-bit integer type. If the compiler does not support 64-bit integers natively, an application or driver can define the following structures to ensure that it has access to this data:

```c
typedef struct{
    SQLUINTEGER dwLowWord;
    SQLINTEGER dwHighWord;
} SQLUBIGINT

typedef struct{
    SQLUINTEGER dwLowWord;
    SQLINTEGER dwHighWord;
} SQLBIGINT
```

These structures should be aligned to an 8-byte boundary because a 64-bit integer is aligned to the 8-byte boundary.

---

**Data Type Identifiers and Descriptors**

The data types listed in the SQL Data Types and C Data Types sections earlier in this appendix are "concise" data types: Each identifier refers to a single data type. There is a one-to-one correspondence between the identifier and the data type. Descriptors, however, do not in all cases use a single value to identify data types. In some cases, they use a "verbose" data type and a type subcode. For all data types except datetime and interval data types, the verbose type identifier is the same as the concise type identifier and the value in SQL_DESC_DATETIME_INTERVAL_CODE is equal to 0. For datetime and interval data types, however, a verbose type (SQL_DATETIME or SQL_INTERVAL) is stored in SQL_DESC_TYPE, a concise type is stored in SQL_DESC_CONCISE_TYPE, and a subcode for each concise type is stored in SQL_DESC_DATETIME_INTERVAL_CODE. Setting one of these fields affects the others. For more information about these fields, see the SQLSetDescField function description.

When the SQL_DESC_TYPE or SQL_DESC_CONCISE_TYPE field is set for some data types, the SQL_DESC_DATETIME_INTERVAL_PRECISION, SQL_DESC_LENGTH, SQL_DESC_PRECISION, and SQL_DESC_SCALE fields are automatically set to default values, as applicable for the data type. For more information, see the description of the SQL_DESC_TYPE field in SQLSetDescField. If any of the default values set is not appropriate, the application should explicitly set the descriptor field through a call to SQLSetDescField.

The following table shows the concise type identifier, verbose type identifier, and type subcode for each datetime and interval SQL and C type identifier. As this table indicates, for datetime and interval data types, the SQL_DESC_TYPE and SQL_DESC_DATETIME_INTERVAL_CODE fields have the same manifest constants both for SQL data types (in implementation descriptors) and for C data types (in application descriptors).

<table>
<thead>
<tr>
<th>Concise SQL type</th>
<th>Concise C type</th>
<th>Verbose type</th>
<th>DATETIME_INTERVAL_CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_TYPE_DATE</td>
<td>SQL_C_TYPE_DATE</td>
<td>SQL_DATETIME</td>
<td>SQL_CODE_DATE</td>
</tr>
<tr>
<td>SQL_TYPE_TIME</td>
<td>SQL_C_TYPE_TIME</td>
<td>SQL_DATETIME</td>
<td>SQL_CODE_TIME</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>SQL_C_TYPE_TIMESTAMP</td>
<td>SQL_DATETIME</td>
<td>SQL_CODE_TIMESTAMP</td>
</tr>
<tr>
<td>SQL_INTERVAL_MONTH</td>
<td>SQL_C_INTERVAL_MONTH</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_MONTH</td>
</tr>
<tr>
<td>SQL_INTERVAL_YEAR</td>
<td>SQL_C_INTERVAL_YEAR</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_YEAR</td>
</tr>
<tr>
<td>SQL_INTERVAL_YEAR_TO_MONTH</td>
<td>SQL_C_INTERVAL_YEAR_TO_MONTH</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_YEAR_TO_MONTH</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY</td>
<td>SQL_C_INTERVAL_DAY</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_DAY</td>
</tr>
<tr>
<td>SQL_INTERVAL_HOUR</td>
<td>SQL_C_INTERVAL_HOUR</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_HOUR</td>
</tr>
<tr>
<td>SQL_INTERVAL_MINUTE</td>
<td>SQL_C_INTERVAL_MINUTE</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_MINUTE</td>
</tr>
<tr>
<td>SQL_INTERVAL_SECOND</td>
<td>SQL_C_INTERVAL_SECOND</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_SECOND</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_HOUR</td>
<td>SQL_C_INTERVAL_DAY_TO_HOUR</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_DAY_TO_HOUR</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_MINUTE</td>
<td>SQL_C_INTERVAL_DAY_TO_MINUTE</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_DAY_TO_MINUTE</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_SECOND</td>
<td>SQL_C_INTERVAL_DAY_TO_SECOND</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_DAY_TO_SECOND</td>
</tr>
<tr>
<td>SQL_INTERVAL_HOUR_TO_MINUTE</td>
<td>SQL_C_INTERVAL_HOUR_TO_MINUTE</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_HOUR_TO_MINUTE</td>
</tr>
<tr>
<td>SQL_INTERVAL_HOUR_TO_SECOND</td>
<td>SQL_C_INTERVAL_HOUR_TO_SECOND</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_HOUR_TO_SECOND</td>
</tr>
<tr>
<td>SQL_INTERVAL_MINUTE_TO_SECOND</td>
<td>SQL_C_INTERVAL_MINUTE_TO_SECOND</td>
<td>SQL_INTERVAL</td>
<td>SQL_CODE_MINUTE_TO_SECOND</td>
</tr>
</tbody>
</table>
Pseudo-Type Identifiers

For application programming convenience, ODBC defines a number of pseudo-type identifiers. These identifiers do not actually correspond to actual data types, but instead, depending on the situation, resolve to existing data types.

This section contains the following topics.

- Default C Data Types
- Bookmark C Data Type
- SQL_ARD_TYPE
- SQL_C_TCHAR

Default C Data Types

If an application specifies SQL_C_DEFAULT in SQLBindCol, SQLGetData, or SQLBindParameter, the driver assumes that the C data type of the output or input buffer corresponds to the SQL data type of the column or parameter to which the buffer is bound.

Important

Interoperable applications should not use SQL_C_DEFAULT. Instead, they should always specify the C type of the buffer they are using. This is because drivers cannot always correctly determine the default C type, for the following reasons:

- If the DBMS promotes an SQL data type of a column or parameter, the driver cannot determine the original SQL data type of a column or parameter. Therefore, it cannot determine the corresponding default C data type.
- If the driver cannot determine whether a particular column or parameter is signed, as is often the case when this is handled by the DBMS, the driver cannot determine whether the corresponding default C data type should be signed or unsigned.

Because SQL_C_DEFAULT is provided only as a programming convenience, the application does not lose any functionality when it specifies the actual C data type.

A table showing the default C data type for each SQL data type is included in Converting Data from SQL to C Data Types, later in this appendix.

Bookmark C Data Type

The bookmark C data type allows an application to retrieve a bookmark. The bookmark C types are used only to retrieve bookmark values that can be variable in length; they should not be converted to other data types. An application retrieves a bookmark either from column 0 of the result set with SQLBulkOperations (with an operation of SQL_ADD), SQLFetch, SQLFetchScroll, or SQLGetData. For more information, see Bookmarks.

The following table lists the value of CType for the bookmark C data type, the ODBC C data type that implements the bookmark C data type, and the definition of this data type from SQL.H.

### Note

The SQL_C_BOOKMARK data type has been deprecated. ODBC 3.x applications should not use SQL_C_BOOKMARK. ODBC 3.x drivers need to support SQL_C_BOOKMARK only if they want to work with ODBC 2.x applications that use it. The Driver Manager maps SQL_C_VARBOOKMARK to SQL_C_BOOKMARK when an application works with an ODBC 2.x driver.

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>ODBC C typedef</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_BOOKMARK (Deprecated)</td>
<td>BOOKMARK</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>SQL_C_VARBOOKMARK</td>
<td>SQLCHAR *</td>
<td>unsigned char *</td>
</tr>
</tbody>
</table>
The SQL_ARD_TYPE type identifier is used to indicate that the data in a buffer will be of the type specified in the SQL_DESC_CONCISE_TYPE field of the ARD. SQL_ARD_TYPE is entered in the TargetType argument of a call to SQLGetData instead of a specific data type and enables an application to change the data type of the buffer by changing the descriptor field. This value ties the data type of the *TargetValuePtr buffer to the descriptor field. (SQL_ARD_TYPE is not entered in a call to SQLBindCol or SQLBindParameter because the type of the bound buffer is already tied to the SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE fields and can be changed at any time by changing either of those fields.)

The SQL_ARD_TYPE type identifier can be used to specify nondefault values for leading precision and seconds precision of interval data types, and precision and scale values for the SQL_C_NUMERIC data type. For more information, see Overriding Default Leading and Seconds Precision for Interval Data Types and Overriding Default Precision and Scale for Numeric Data Types, later in this appendix.

The SQL_C_TCHAR type identifier does not actually identify a data type; it is a macro that exists within the header file for Unicode conversion. It is replaced by SQL_C_CHAR or SQL_C_WCHAR depending on the setting of the UNICODE #define. It is useful for an application transferring character data that is compiled as both an ANSI and a Unicode application.

An application can safely transfer data (in the internal form used by a specified DBMS) between two data sources that use the same DBMS and hardware platform. For a given piece of data, the SQL data types must be the same in the source and target data sources. The C data type is SQL_C_BINARY.

When the application calls SQLFetch, SQLFetchScroll, or SQLGetData to retrieve the data from the source data source, the driver retrieves the data from the data source and transfers it, without conversion, to a storage location of type SQL_C_BINARY. When the application calls SQLBulkOperations, SQLExecute, SQLExecDirect, SQLPutData, or SQLSetPos to send the data to the target data source, the driver retrieves the data from the storage location and transfers it, without conversion, to the target data source.

Applications that transfer any data (except binary data) in this manner are not interoperable among DBMSs.

SQLCopyDesc can be used to copy row bindings from the source DBMS to parameter bindings in the target DBMS.

Guidelines for Interval and Numeric Data Types

The following sections address interval and numeric data types.

- Interval Data Types
- Numeric Literals
- Overriding Default Precision and Scale for Numeric Data Types
Interval Data Types

An interval is defined as the difference between two dates and times. Intervals are expressed in one of two different ways. One is a year-month interval that expresses intervals in terms of years and an integral number of months. The other is a day-time interval that expresses intervals in terms of days, minutes, and seconds. These two types of intervals are distinct and cannot be mixed, because months can have varying numbers of days.

An interval consists of a set of fields. There is an implied ordering among the fields. For example, in a year-to-month interval, the year comes first, followed by the month. Similarly, in a day-to-minute interval, the fields are in the order day, hour, and minute. The first field in an interval type is called the leading field, or the high-order field. The last field is called the trailing field.

In all intervals, the leading field is not constrained by rules of the Gregorian calendar. For example, in an hour-to-minute interval, the hour field is not constrained to be between 0 and 23 (inclusive), as it normally is. The trailing fields subsequent to the leading field follow the usual constraints of the Gregorian calendar. For more information, see Constraints of the Gregorian Calendar, later in this appendix.

There are 13 interval SQL data types and 13 interval C data types. Each of the interval C data types uses the same structure, SQL_INTERVAL_STRUCT, to contain the interval data. (For more information, see the next section, C Interval Structure.) For more information on the SQL data types, see SQL Data Types; for more information on the C data types, see C Data Types.

<table>
<thead>
<tr>
<th>Type identifier</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONTH</td>
<td>Year-Month</td>
<td>Number of months between two dates.</td>
</tr>
<tr>
<td>YEAR</td>
<td>Year-Month</td>
<td>Number of years between two dates.</td>
</tr>
<tr>
<td>YEAR_TO_MONTH</td>
<td>Year-Month</td>
<td>Number of years and months between two dates.</td>
</tr>
<tr>
<td>DAY</td>
<td>Day-Time</td>
<td>Number of days between two dates.</td>
</tr>
<tr>
<td>HOUR</td>
<td>Day-Time</td>
<td>Number of hours between two date/times.</td>
</tr>
<tr>
<td>MINUTE</td>
<td>Day-Time</td>
<td>Number of minutes between two date/times.</td>
</tr>
<tr>
<td>SECOND</td>
<td>Day-Time</td>
<td>Number of seconds between two date/times.</td>
</tr>
<tr>
<td>DAY_TO_HOUR</td>
<td>Day-Time</td>
<td>Number of days/hours between two date/times.</td>
</tr>
<tr>
<td>DAY_TO_SECOND</td>
<td>Day-Time</td>
<td>Number of days/hours/minutes between two date/times.</td>
</tr>
<tr>
<td>HOUR_TO_MINUTE</td>
<td>Day-Time</td>
<td>Number of hours/minutes/seconds between two date/times.</td>
</tr>
<tr>
<td>HOUR_TO_SECOND</td>
<td>Day-Time</td>
<td>Number of hours/minutes/seconds between two date/times.</td>
</tr>
<tr>
<td>MINUTE_TO_SECOND</td>
<td>Day-Time</td>
<td>Number of minutes/seconds between two date/times.</td>
</tr>
</tbody>
</table>

This section contains the following topics.

- C Interval Structure
- Interval Data Type Precision
- Interval Data Type Length
- Interval Literals
- Overriding Default Leading and Seconds Precision for Interval Data Types

© 2016 Microsoft

C Interval Structure

Each of the C interval data types listed in the C Data Types section uses the same structure to contain the interval data. When SQLFetch, SQLFetchScroll, or SQLGetData is called, the driver returns data into the SQL_INTERVAL_STRUCT structure, uses the value that was specified by the application for the C data types (in the call to SQLBindCol, SQLGetData, or SQLBindParameter) to interpret the contents of SQL_INTERVAL_STRUCT, and populates the interval_type field of the structure with the enum value corresponding to the C type. Note that drivers do not read the interval_type field to determine the type of the interval; they retrieve the value of the SQL_DESC_CONCISE_TYPE descriptor field. When the structure is used for parameter data, the driver uses the value specified by the application in the SQL_DESC_CONCISE_TYPE field of the APD to interpret the contents of SQL_INTERVAL_STRUCT, even if the application sets the value of the interval_type field to a different value.
This structure is defined as follows:

```c
typedef struct tagSQL_INTERVAL_STRUCT
{
    SQLINTERVAL interval_type;
    SQLSMALLINT interval_sign;
    union
    {
        SQL_YEAR_MONTH_STRUCT year_month;
        SQL_DAY_SECOND_STRUCT day_second;
    } intval;
} SQL_INTERVAL_STRUCT;

typedef enum
{
    SQL_IS_YEAR = 1,
    SQL_IS_MONTH = 2,
    SQL_IS_DAY = 3,
    SQL_IS_HOUR = 4,
    SQL_IS_MINUTE = 5,
    SQL_IS_SECOND = 6,
    SQL_IS_YEAR_TO_MONTH = 7,
    SQL_IS_DAY_TO_HOUR = 8,
    SQL_IS_DAY_TO_MINUTE = 9,
    SQL_IS_DAY_TO_SECOND = 10,
    SQL_IS_HOUR_TO_MINUTE = 11,
    SQL_IS_HOUR_TO_SECOND = 12,
    SQL_IS_MINUTE_TO_SECOND = 13
} SQLINTERVAL;

typedef struct tagSQL_YEAR_MONTH
{
    SQLUINTEGER year;
    SQLUINTEGER month;
} SQL_YEAR_MONTH_STRUCT;

typedef struct tagSQL_DAY_SECOND
{
    SQLUINTEGER day;
    SQLUINTEGER hour;
    SQLUINTEGER minute;
    SQLUINTEGER second;
    SQLUINTEGER fraction;
} SQL_DAY_SECOND_STRUCT;
```

The interval_type field of the SQL_INTERVAL_STRUCT indicates to the application what structure is held in the union and also what members of the structure are relevant. The interval_sign field has the value SQL_FALSE if the interval leading field is unsigned; if it is SQL_TRUE, the leading field is negative. The value in the leading field itself is always unsigned, regardless of the value of interval_sign. The interval_sign field acts as a sign bit.

**Interval Data Type Precision**

Precision for an interval data type includes interval leading precision, interval precision, and seconds precision.

The leading field of an interval is a signed numeric. The maximum number of digits for the leading field is determined by a quantity called interval leading precision, which is a part of the data type declaration. For example, the declaration: INTERVAL HOUR(5) TO MINUTE has an interval leading precision of 5; the HOUR field can take values from –99999 through 99999. The interval leading precision is contained in the SQL_DESC_DATETIME_INTERVAL_PRECISION field of the descriptor record.

The list of fields that an interval data type is made up of is called interval precision. It is not a numeric value, as the term "precision" might imply. For example, the interval precision of the type INTERVAL DAY TO SECOND is the list DAY, HOUR, MINUTE, SECOND. There is no descriptor field that holds this value; the interval precision can always be determined by the interval data type.

Any interval data type that has a SECOND field has a seconds precision. This is the number of decimal digits allowed in the fractional part of the seconds value. This is different than for other data types, where precision indicates the number of digits before the decimal point. The seconds precision of an interval data type is the number of digits after the decimal point. For example, if the seconds precision is set to 6, the number 123456 in the fraction field would be interpreted as 123456 and the number 1230 would be interpreted as .001230. For other data types, this is referred to as scale. Interval seconds precision is contained in the SQL_DESC_PRECISION field of the descriptor. If the precision of the fractional seconds component of the SQL interval value is greater than what can be held in the C interval structure, it is driver-defined whether the fractional seconds value in the SQL interval is rounded or truncated when converted to the C interval structure.

When the SQL_DESC_CONCISE_TYPE field is set to an interval data type, the SQL_DESC_TYPE field is set to SQL_INTERVAL and the SQL_DESC_DATETIME_INTERVAL_CODE is set to the code for the interval data type. The SQL_DESC_DATETIME_INTERVAL_PRECISION field is automatically set to the default interval leading precision of 2, and the SQL_DESC_PRECISION field is automatically set to the default interval seconds precision of 6. If either of these values is not appropriate, the application should explicitly set the descriptor field through a call to SQLSetDescField.
Interval Data Type Length

The following rules are used to determine the length of an interval data type in characters. Length is expressed in number of characters. The number of bytes depends upon the character set. The length includes the following values added together:

- Two characters for every field in the interval that is not the leading field.
- For the leading field, the number of characters that is the express or implicit leading precision. If the leading precision is not specified, the default value is 2.
- One character for the separator between the fields.
- One plus the express or implied seconds precision. If the seconds precision is not specified, the default value is 6.

Specific column length values for each interval data type are contained in Column Size.

Interval Literals

ODBC requires that all drivers support conversion of the SQL_CHAR or SQL_VARCHAR data type to all C interval data types. If the underlying data source does not support interval data types, however, the driver needs to know the correct format of the value in the SQL_CHAR field in order to support these conversions. Similarly, ODBC requires that any ODBC C type be convertible to SQL_CHAR or SQL_VARCHAR, so a driver needs to know what format an interval stored in the character field should have. This section describes the syntax of interval literals, which the driver writer needs to use to validate the SQL_CHAR fields during conversion either to or from C interval data types.

**Note**
The complete BNF syntax for interval literals is shown in the section Interval Literal Syntax in Appendix C: SQL Grammar.

To pass interval literals as part of an SQL statement, an escape clause syntax is defined for interval literals. For more information, see Date, Time, and Timestamp Literals.

An interval literal is of the form:

```
INTERVAL[<sign>] 'value' <interval qualifier>
```

where "INTERVAL" indicates that the character literal is an interval. The sign can be either plus or minus; it is outside the interval string and is optional.

The interval qualifier can either be a single datetime field or be composed of two datetime fields, in the form: `<leading field> TO <trailing field>`.

- When the interval is composed of a single field, the single field can be a non-second field that may be accompanied by an optional leading precision in parentheses. The single datetime field can also be a second field that may be accompanied by the optional leading precision, the optional fractional-seconds precision in parentheses, or both. If both a leading precision and a fractional-seconds precision are present for a seconds field, they are separated by commas. If the seconds field has a fractional-seconds precision, it must also have a leading precision.
- When the interval is composed of leading and trailing fields, the leading field is a non-second field that may be accompanied by the interval leading field precision in parentheses. The trailing field can be either a non-second field or a second field that may be accompanied by an interval fractional-seconds precision in parentheses.

The interval string in value is enclosed in single quotation marks. It can be either a year-month literal or a day-time literal. The format of the string in value is determined by the following rules:

- The string contains a decimal value for every field that is implied by the <interval qualifier>.
- If the interval precision includes the fields YEAR and MONTH, the values of these fields are separated by a minus sign.
- If the interval precision includes the fields DAY and HOUR, the values of these fields are separated by a space.
- If the interval precision includes the field HOUR and the lower order fields (MINUTE and SECOND), the values of these fields are separated by a colon.
- If the interval precision includes a SECOND field and the expressed or implied seconds precision is nonzero, the character immediately before the first digit of the fractional part of the second is a period.
- No field can be more than two digits long, except:
  - The value of the leading field can be as long as the expressed or implied interval leading precision.
  - The fractional part of the SECOND field can be as long as the expressed or implied seconds precision.
  - The trailing fields follow the usual constraints of the Gregorian calendar. (See Constraints of the Gregorian Calendar.)
The following table lists examples of valid interval literals as included in the ODBC escape clause for intervals. The syntax of the escape clause is as follows:

\( \text{INTERVAL sign interval-string interval-qualifier} \)

<table>
<thead>
<tr>
<th>Literal escape clause</th>
<th>Interval specified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{INTERVAL '326' YEAR(4)}</code></td>
<td>Specifies an interval of 326 years. The interval leading precision is 4.</td>
</tr>
<tr>
<td><code>{INTERVAL '326' MONTH(3)}</code></td>
<td>Specifies an interval of 326 months. The interval leading precision is 3.</td>
</tr>
<tr>
<td><code>{INTERVAL '3261' DAY(4)}</code></td>
<td>Specifies an interval of 3261 days. The interval leading precision is 4.</td>
</tr>
<tr>
<td><code>{INTERVAL '163' HOUR(3)}</code></td>
<td>Specifies an interval of 163 days. The interval leading precision is 3.</td>
</tr>
<tr>
<td><code>{INTERVAL '163' MINUTE(3)}</code></td>
<td>Specifies an interval of 163 minutes. The interval leading precision is 3.</td>
</tr>
<tr>
<td><code>{INTERVAL '223.16' SECOND(3,2)}</code></td>
<td>Specifies an interval of 223.16 seconds. The interval leading precision is 3, and the seconds precision is 2.</td>
</tr>
<tr>
<td><code>{INTERVAL '163-11' YEAR(3) TO MONTH)</code></td>
<td>Specifies an interval of 163 years and 11 months. The interval leading precision is 3.</td>
</tr>
<tr>
<td><code>{INTERVAL '163 12' DAY(3) TO HOUR)</code></td>
<td>Specifies an interval of 163 days and 12 hours. The interval leading precision is 3.</td>
</tr>
<tr>
<td><code>{INTERVAL '163 12:39' DAY(3) TO MINUTE)</code></td>
<td>Specifies an interval of 163 days, 12 hours, and 39 minutes. The interval leading precision is 3.</td>
</tr>
<tr>
<td><code>{INTERVAL '163 12:39:59.163' DAY(3) TO SECOND(3))}</code></td>
<td>Specifies an interval of 163 days, 12 hours, 39 minutes, and 59.163 seconds. The interval leading precision is 3, and the seconds precision is 3.</td>
</tr>
<tr>
<td><code>{INTERVAL '163:39' HOUR(3) TO MINUTE)</code></td>
<td>Specifies an interval of 163 hours and 39 minutes. The interval leading precision is 3.</td>
</tr>
<tr>
<td><code>{INTERVAL '163:39:59.163' HOUR(3) TO SECOND(4))}</code></td>
<td>Specifies an interval of 163 hours, 39 minutes, and 59.163 seconds. The interval leading precision is 3, and the seconds precision is 4.</td>
</tr>
<tr>
<td><code>{INTERVAL '163:59.163' MINUTE(3) TO SECOND(5))}</code></td>
<td>Specifies an interval of 163 minutes and 59.163 seconds. The interval leading precision is 3, and the seconds precision is 5.</td>
</tr>
<tr>
<td><code>{INTERVAL '16-23:39:56.23' DAY TO SECOND)}</code></td>
<td>Specifies an interval of minus 16 days, 23 hours, 39 minutes, and 56.23 seconds. The implied leading precision is 2, and the implied seconds precision is 6.</td>
</tr>
</tbody>
</table>

The following table lists examples of invalid interval literals:

<table>
<thead>
<tr>
<th>Literal escape clause</th>
<th>Reason why invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{INTERVAL '163' HOUR(2)}</code></td>
<td>The interval leading precision is 2, but the value of the leading field is 163.</td>
</tr>
<tr>
<td><code>{INTERVAL '223.16' SECOND(2,2)}</code></td>
<td>In the first example, the leading precision is too small, and in the second example, the seconds precision is too small.</td>
</tr>
<tr>
<td><code>{INTERVAL '223.16' SECOND(3,1)}</code></td>
<td>Because the leading precision is unspecified, it defaults to 2, which is too small to hold the specified literal.</td>
</tr>
<tr>
<td><code>{INTERVAL '22.1234567' SECOND)}</code></td>
<td>The seconds precision is unspecified, so it defaults to 6. The literal has seven digits after the decimal point.</td>
</tr>
<tr>
<td><code>{INTERVAL '163-11' YEAR(3) TO MONTH)</code></td>
<td>The trailing field does not follow the rules of the Gregorian calendar.</td>
</tr>
<tr>
<td><code>{INTERVAL '163 65' DAY(3) TO HOUR)</code></td>
<td></td>
</tr>
<tr>
<td><code>{INTERVAL '163 62:39' DAY(3) TO MINUTE)</code></td>
<td></td>
</tr>
<tr>
<td><code>{INTERVAL '163 12:125:59.163' DAY(3) TO SECOND(3))}</code></td>
<td></td>
</tr>
<tr>
<td><code>{INTERVAL '163:144' HOUR(3) TO MINUTE)</code></td>
<td></td>
</tr>
<tr>
<td><code>{INTERVAL '163:567:234.163' HOUR(3) TO SECOND(4))}</code></td>
<td></td>
</tr>
</tbody>
</table>
Overriding Default Leading and Seconds Precision for Interval Data Types

When the SQL_DESC_TYPE field of an ARD is set to a datetime or interval C type, by calling either SQLBindCol or SQLSetDescField, the SQL_DESC_PRECISION field (which contains the interval seconds precision) is set to the following defaults:

- 6 for timestamp and all interval data types with a second component.
- 0 for all other data types.

For all interval data types, the SQL_DESC_DATETIME_INTERVAL_PRECISION descriptor field, which contains the interval leading field precision, is set to a default value of 2.

When the SQL_DESC_TYPE field in an APD is set to a datetime or interval C type, by calling either SQLBindParameter or SQLSetDescField, the SQL_DESC_PRECISION and SQL_DESC_DATETIME_INTERVAL_PRECISION fields in the APD are set to the default given previously. This is true for input parameters but not for input/output or output parameters.

A call to SQLSetDescRec sets the interval leading precision to the default but sets the interval seconds precision (in the SQL_DESC_PRECISION field) to the value of its Precision argument.

If either of the defaults given previously is not acceptable to an application, the application should set the SQL_DESC_PRECISION or SQL_DESC_DATETIME_INTERVAL_PRECISION field by calling SQLSetDescField.

If the application calls SQLGetData to return data into a datetime or interval C type, the default interval leading precision and interval seconds precision are used. If either default is not acceptable, the application must call SQLSetDescField to set either descriptor field, or SQLSetDescRec to set SQL_DESC_PRECISION. The call to SQLGetData should have a TargetType of SQL_ARD_TYPE to use the values in the descriptor fields.

When SQLPutData is called, the interval leading precision and interval seconds precision are read from the fields of the descriptor record that correspond to the data-at-execution parameter or column, which are APD fields for calls to SQLExecute or SQLExecDirect, or ARD fields for calls to SQLBulkOperations or SQLSetPos.

Numeric Literals

Numeric literals are used when numeric data values are stored in character strings. For conversion of numeric SQL data to an SQL_C_CHAR string, or conversion of numeric C data to an SQL_CHAR or SQL_VARCHAR string, numeric literal syntax is used to specify what is stored in the target. For conversion of a numeric stored as a SQL_C_CHAR string to numeric SQL data, or a numeric stored as a SQL_CHAR string to numeric C data, this syntax is used to validate what is stored in the source.

Numeric literals should conform to the syntax defined in the section Numeric Literal Syntax in Appendix C: SQL Grammar.

This section contains the following topics.

- Rules for Conversions
- Overriding Default Precision and Scale for Numeric Data Types

Rules for Conversions

The rules in this section apply for conversions involving numeric literals. For the purposes of these rules, the following terms are defined:

- **Store assignment**: When sending data into a table column in a database. This occurs during calls to SQLExecute, SQLExecDirect, and SQLSetPos. During store assignment, "target" refers to a database column and "source" refers to data in application buffers.
- **Retrieval assignment**: When retrieving data from the database into application buffers. This occurs during calls to SQLFetch, SQLGetData, SQLFetchScroll, and SQLSetPos. During retrieval assignment, "target" refers to the application buffers and "source" refers to the database column.
- **CS**: The value in the character source.
- **NT**: The value in the numeric target.
- **NS**: The value in the numeric source.
- **CT**: The value in the character target.
- Precision of an exact numeric literal: the number of digits it contains.
- The scale of an exact numeric literal: the number of digits to the right of the expressed or implied period.
- The precision of an approximate numeric literal: the precision of its mantissa.

### Character Source to Numeric Target

Following are the rules for converting from a character source (CS) to a numeric target (NT):

1. Replace CS with the value obtained by removing any leading or trailing spaces in CS. If CS is not a valid numeric-literal, SQLSTATE 22018 (Invalid character value for cast specification) is returned.
2. Replace CS with the value obtained by removing leading zeroes before the decimal point, trailing zeroes after the decimal point, or both.
3. Convert CS to NT. If the conversion results in a loss of significant digits, SQLSTATE 22003 (Numeric value out of range) is returned. If the conversion results in the loss of nonsignificant digits, SQLSTATE 01S07 (Fractional truncation) is returned.

### Numeric Source to Character Target

Following are the rules for converting from a numeric source (NS) to a character target (CT):

1. Let LT be the length in characters of CT. For retrieval assignment, LT is equal to the length of the buffer in characters minus the number of bytes in the null-termination character for this character set.
2. Cases:
   - If NS is an exact numeric type, then let YP equal the shortest character string that conforms to the definition of exact-numeric-literal such that the scale of YP is the same as the scale of NS, and the interpreted value of YP is the absolute value of NS.
   - If NS is an approximate numeric type, then let YP be a character string as follows:
     
     Case:
     
     If NS is equal to 0, then YP is 0.
     
     Let YSN be the shortest character string that conforms to the definition of exact-numeric-literal and whose interpreted value is the absolute value of NS. If the length of YSN is less than the \( (\text{precision} + 1) \) of the data type of NS, then let YP equal YSN.
     
     Otherwise, YP is the shortest character string that conforms to the definition of approximate-numeric-literal whose interpreted value is the absolute value of NS and whose mantissa consists of a single digit that is not '0', followed by a period and an unsigned-integer.
   
   3. Case:
      - If NS is less than 0, then let Y be the result of:
        
        `' '-' || YP`
        
        where `'||'` is the string concatenation operator.
        
        Otherwise, let Y equal YP.
      
        4. Let LY be the length in characters of Y.
      
      5. Case:
         - If LY equals LT, then CT is set to Y.
         - If LY is less than LT, then CT is set to Y extended on the right by appropriate number of spaces.
         
         Otherwise (LY > LT), copy the first LT characters of Y into CT.
         
         Case:
         
         If this is a store assignment, return the error SQLSTATE 22001 (String data, right-truncated).
         
         If this is retrieval assignment, return the warning SQLSTATE 01004 (String data, right-truncated). When the copy results in the loss of fractional digits (other than trailing zeros), it is driver-defined whether one of the following occurs:
         
         1. The driver truncates the string in Y to an appropriate scale (which can be zero also) and writes the result into CT.
         2. The driver rounds the string in Y to an appropriate scale (which can be zero also) and writes the result into CT.
         3. The driver neither truncates nor rounds, but just copies the first LT characters of Y into CT.
Overriding Default Precision and Scale for Numeric Data Types

When the SQL_DESC_TYPE field in an ARD is set to SQL_C_NUMERIC, by calling either SQLBindCol or SQLSetDescField, the SQL_DESC_SCALE field in the ARD is set to 0 and the SQL_DESC_PRECISION field is set to a driver-defined default precision. This is also true when the SQL_DESC_TYPE field in an APD is set to SQL_C_NUMERIC, by calling either SQLBindParameter or SQLSetDescField. This is true for input, input/output, or output parameters.

If either of the defaults described previously are not acceptable for an application, the application should set the SQL_DESC_SCALE or SQL_DESC_PRECISION field by calling SQLSetDescField or SQLSetDescRec.

If the application calls SQLGetData to return data into an SQL_C_NUMERIC structure, the default SQL_DESC_SCALE and SQL_DESC_PRECISION fields are used. If the defaults are not acceptable, the application must call SQLSetDescRec or SQLSetDescField to set the fields and then call SQLGetData with a TargetType of SQL_ARC_TYPE to use the values in the descriptor fields.

When SQLPutData is called, the call uses the SQL_DESC_SCALE and SQL_DESC_PRECISION fields of the descriptor record that corresponds to the data-at-execution parameter or column, which are APD fields for calls to SQLExecute or SQLExecDirect, or ARD fields for calls to SQLBulkOperations or SQLSetPos.

Constraints of the Gregorian Calendar

Date and datetime data types, and the trailing fields of interval data types, must conform to the constraints of the Gregorian calendar. These constraints are as follows:

- The value of the month field must be between 1 and 12, inclusive.
- The value of the day field must be in the range from 1 through the number of days in the month. The number of days in the month is determined from the values of the year and months fields and can be 28, 29, 30, or 31. (The number of days in the month can also depend on whether it is a leap year.)
- The value of the hour field must be between 0 and 23, inclusive.
- The value of the minute field must be between 0 and 59, inclusive.
- For the trailing seconds field of interval data types, the value of the seconds field must be between 0 and 59.9(n), inclusive, where n is the number of digits in the fractional seconds precision.
- For the trailing seconds field of datetime data types, the value of the seconds field must be between 0 and 61.9(n), inclusive, where n specifies the number of "9" digits and the value of n is the fractional seconds precision. (The range of seconds allows as many as two leap seconds to maintain synchronization of sidereal time.)

Column Size, Decimal Digits, Transfer Octet Length, and Display Size

Data types are characterized by their column (or parameter) size, decimal digits, length, and display size. The following ODBC functions return these attributes for a parameter in an SQL statement or for an SQL data type on a data source. Each ODBC function returns a different set of these attributes, as follows:

- SQLDescribeCol returns the column size and decimal digits of the columns it describes.
- SQLDescribeParam returns the parameter size and decimal digits of the parameters it describes. SQLBindParameter sets the parameter size and decimal digits for a parameter in an SQL statement.
- The catalog functions SQLColumns, SQLProcedureColumns, and SQLGetTypeInfo return attributes for a column in a table, result set, or a procedure parameter and the catalog attributes of the data types in the data source. SQLColumns returns the column size, decimal digits, and length of a column in specified tables (such as the base table, view, or a system table). SQLProcedureColumns returns the column size, decimal digits, and length of a column in a procedure. SQLGetTypeInfo returns the maximum column size and the minimum and maximum decimal digits of an SQL data type on a data source.

The values returned by these functions for the column or parameter size correspond to "precision" as defined in ODBC 2.x. However, the values do not necessarily correspond to the values returned in SQL_DESC_PRECISION or any other one descriptor field. The same is true for decimal digits, which correspond to "scale" as defined in ODBC 2.x. It does not necessarily correspond to the values returned in SQL_DESC_SCALE or any other one descriptor field, but comes from different descriptor fields depending on the data type. For further information, see Column Size and Decimal Digits.

Similarly, the values for transfer octet length do not come from SQL_DESC_LENGTH. They come from the SQL_DESC_OCTET_LENGTH of a field of a descriptor for all character and binary types. There is no descriptor field that holds this information for other types.
The display size value for all data types corresponds to the value in a single descriptor field, SQL_DESC_DISPLAY_SIZE.

Descriptor fields describe the characteristics of a result set. Descriptor fields do not contain valid values about data before statement execution. The values for column size, decimal digits, and display size returned by SQLColumns, SQLProcedureColumns, and SQLGetTypeInfo, on the other hand, return characteristics of database objects, such as table columns and data types, that exist in the data source's catalog. Likewise, in its result set, SQLColAttribute returns the column size, decimal digits, and transfer octet length of columns at the data source; these values are not necessarily the same as the values in the SQL_DESC_PRECISION, SQL_DESC_SCALE, and SQL_DESC_OCTET_LENGTH descriptor fields.

For more information about these descriptor fields, see SQLSetDescField.

This section contains the following topics:

- Column Size
- Decimal Digits
- Transfer Octet Length
- Display Size

© 2016 Microsoft

## Column Size

The column (or parameter) size of numeric data types is defined as the maximum number of digits used by the data type of the column or parameter, or the precision of the data. For character types, this is the length in characters of the data; for binary data types, column size is defined as the length in bytes of the data. For the time, timestamp, and all interval data types, this is the number of characters in the character representation of this data. The column size defined for each concise SQL data type is shown in the following table.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Column size</th>
</tr>
</thead>
<tbody>
<tr>
<td>All character types</td>
<td>The defined or maximum column size in characters of the column or parameter (as contained in the SQL_DESC_LENGTH descriptor field). For example, the column size of a single-byte character column defined as CHAR(10) is 10.</td>
</tr>
<tr>
<td>SQL_DECIMAL SQL_NUMERIC</td>
<td>The defined number of digits. For example, the precision of a column defined as NUMERIC(10,3) is 10.</td>
</tr>
<tr>
<td>SQL_BIT[c]</td>
<td>1</td>
</tr>
<tr>
<td>SQL_TINYINT[c]</td>
<td>3</td>
</tr>
<tr>
<td>SQL_SMALLINT[c]</td>
<td>5</td>
</tr>
<tr>
<td>SQL_INTEGER[c]</td>
<td>10</td>
</tr>
<tr>
<td>SQL_BIGINT[c]</td>
<td>19 (if signed) or 20 (if unsigned)</td>
</tr>
<tr>
<td>SQL_REAL[c]</td>
<td>7</td>
</tr>
<tr>
<td>SQL_FLOAT[c]</td>
<td>15</td>
</tr>
<tr>
<td>SQL_DOUBLE[c]</td>
<td>15</td>
</tr>
<tr>
<td>All binary types</td>
<td>The defined or maximum length in bytes of the column or parameter. For example, the length of a column defined as BINARY(10) is 10.</td>
</tr>
<tr>
<td>SQL_TYPE_DATE[c]</td>
<td>10 (the number of characters in the yyyy-mm-dd format).</td>
</tr>
<tr>
<td>SQL_TYPE_TIME[c]</td>
<td>8 (the number of characters in the hh-mm-ss format), or 9 + s (the number of characters in the hh:mm:ss[.fff...] format, where s is the seconds precision).</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>16 (the number of characters in the yyyy-mm-dd hh:mm format)</td>
</tr>
<tr>
<td></td>
<td>19 (the number of characters in the yyyy-mm-dd hh:mm:ss format) or 20 + s (the number of characters in the yyyy-mm-dd hh:mm:ss[.fff...] format, where s is the seconds precision).</td>
</tr>
<tr>
<td>SQL_INTERVAL_SECOND</td>
<td>Where p is the interval leading precision and s is the seconds precision, p (if s=0) or p+s+1 (if s&gt;0).</td>
</tr>
</tbody>
</table>

[d]
SQL_INTERVAL_DAY_TO_SECOND
Where \( p \) is the interval leading precision and \( s \) is the seconds precision, \( 9+p \) (if \( s=0 \)) or \( 10+p+s \) (if \( s>0 \)).[d]

SQL_INTERVAL_HOUR_TO_SECOND
Where \( p \) is the interval leading precision and \( s \) is the seconds precision, \( 6+p \) (if \( s=0 \)) or \( 7+p+s \) (if \( s>0 \)).[d]

SQL_INTERVAL_MINUTE_TO_SECOND
Where \( p \) is the interval leading precision and \( s \) is the seconds precision, \( 3+p \) (if \( s=0 \)) or \( 4+p+s \) (if \( s>0 \)).[d]

SQL_INTERVAL_YEAR SQL_INTERVAL_MONTH SQL_INTERVAL_DAY SQL_INTERVAL_HOUR SQL_INTERVAL_MINUTE
\( p \), where \( p \) is the interval leading precision.[d]

SQL_INTERVAL_YEAR_TO_MONTH SQL_INTERVAL_DAY_TO_HOUR SQL_INTERVAL_DAY_TO_MINUTE
\( 3+p \), where \( p \) is the interval leading precision.[d]

SQL_INTERVAL_DAY_TO_MINUTE
\( 6+p \), where \( p \) is the interval leading precision.[d]

SQL_INTERVAL_HOUR_TO_MINUTE
\( 3+p \), where \( p \) is the interval leading precision.[d]

SQL_GUID
36 (the number of characters in the `aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee` format)

[a] For an ODBC 1.0 application calling `SQLSetParam` in an ODBC 2.0 driver, and for an ODBC 2.0 application calling `SQLBindParameter` in an ODBC 1.0 driver, when \(*StrLen_or_IndPtr\) is SQL_DATA_AT_EXEC for a SQL_LONGVARCHAR or SQL_LONGVARBINARY type, `ColumnSize` must be set to the total length of the data to be sent, not the precision as defined in this table.

[b] If the driver cannot determine the column or parameter length for a variable type, it returns SQL_NO_TOTAL.

[c] The `ColumnSize` argument of `SQLBindParameter` is ignored for this data type.

[d] For general rules about column length in interval data types, see `Interval Data Type Length`, earlier in this appendix.

The values returned for the column (or parameter) size do not correspond to the values in any one descriptor field. The values can come from either the `SQL_DESC_PRECISION` or the `SQL_DESC_LENGTH` field, depending on the type of data, as shown in the following table.

<table>
<thead>
<tr>
<th>SQL type</th>
<th>Descriptor field corresponding to column or parameter size</th>
</tr>
</thead>
<tbody>
<tr>
<td>All character and binary types</td>
<td>LENGTH</td>
</tr>
<tr>
<td>All numeric types</td>
<td>PRECISION</td>
</tr>
<tr>
<td>All datetime and interval types</td>
<td>LENGTH</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>LENGTH</td>
</tr>
</tbody>
</table>

**Decimal Digits**

The decimal digits of decimal and numeric data types is defined as the maximum number of digits to the right of the decimal point, or the scale of the data. For approximate floating-point number columns or parameters, the scale is undefined because the number of digits to the right of the decimal point is not fixed. For datetime or interval data that contains a seconds component, the decimal digits is defined as the number of digits to the right of the decimal point in the seconds component of the data.

For the SQL_DECIMAL and SQL_NUMERIC data types, the maximum scale is usually the same as the maximum precision. However, some data sources impose a separate limit on the maximum scale. To determine the minimum and maximum scales allowed for a data type, an application calls `SQLGetTypeInfo`.

The decimal digits defined for each concise SQL data type is shown in the following table.

<table>
<thead>
<tr>
<th>SQL type</th>
<th>Decimal digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>All character and binary types[a]</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_DECIMAL SQL_NUMERIC</td>
<td>The defined number of digits to the right of the decimal point. For example, the scale of a column defined as NUMERIC(10,3) is 3. This can be a negative number to support storage of very large numbers without using exponential notation; for example, “12000” could be stored as “12” with a scale of –3.</td>
</tr>
<tr>
<td>All exact numeric types other than SQL_DECIMAL and SQL_NUMERIC[a]</td>
<td>0</td>
</tr>
</tbody>
</table>
All approximate data types
SQL_TYPE_DATE, and all interval types with no seconds component
All datetime types except SQL_TYPE_DATE, and all interval types with a seconds component
SQL_GUID

[a] The Decimals argument of SQLBindParameter is ignored for this data type.

The values returned for the decimal digits do not correspond to the values in any one descriptor field. The values can come from either the SQL_DESC_SCALE or the SQL_DESC_PRECISION field, depending on the data type, as shown in the following table.

<table>
<thead>
<tr>
<th>SQL type</th>
<th>Descriptor field corresponding to decimal digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>All character and binary types</td>
<td>n/a</td>
</tr>
<tr>
<td>All exact numeric types</td>
<td>SCALE</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>n/a</td>
</tr>
<tr>
<td>All approximate numeric types</td>
<td>n/a</td>
</tr>
<tr>
<td>All datetime types</td>
<td>PRECISION</td>
</tr>
<tr>
<td>All interval types with a seconds component</td>
<td>PRECISION</td>
</tr>
<tr>
<td>All interval types with no seconds component</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Transfer Octet Length

The transfer octet length of a column is the maximum number of bytes returned to the application when data is transferred to its default C data type. For character data, the transfer octet length does not include space for the null-termination character. The transfer octet length of a column may be different than the number of bytes required to store the data on the data source.

The transfer octet length defined for each ODBC SQL data type is shown in the following table.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>All character types[a]</td>
<td>The defined or the maximum (for variable type) length of the column in bytes. This is the same value as the descriptor field SQL_DESC_OCTET_LENGTH.</td>
</tr>
<tr>
<td>SQL_DECIMAL SQL_NUMERIC</td>
<td>The number of bytes required to hold the character representation of this data if the character set is ANSI, and twice this number if the character set is UNICODE. This is the maximum number of digits plus two, because the data is returned as a character string and characters are needed for the digits, a sign, and a decimal point. For example, the transfer length of a column defined as NUMERIC(10,3) is 12.</td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td>1</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>2</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>SQL_BIGINT</td>
<td>The number of bytes required to hold the character representation of this data if the character set is ANSI, and twice this number if the character set is UNICODE, because this data type is returned as a character string by default. The character representation consists of 20 characters: 19 for digits and a sign, if signed, or 20 digits, if unsigned. Therefore, the length is 20.</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>4</td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td>8</td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>8</td>
</tr>
</tbody>
</table>
Display Size

The display size of a column is the maximum number of characters needed to display data in character form. The following table defines the display size for each ODBC SQL data type.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Display size</th>
</tr>
</thead>
<tbody>
<tr>
<td>All character types[a]</td>
<td>The defined (for fixed types) or maximum (for variable types) number of characters needed to display the data in character form.</td>
</tr>
<tr>
<td>SQL_DECIMAL, SQL_NUMERIC</td>
<td>The precision of the column plus 2 (a sign, precision digits, and a decimal point). For example, the display size of a column defined as NUMERIC(10,3) is 12.</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>1 (1 digit).</td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td>4 if signed (a sign and 3 digits) or 3 if unsigned (3 digits).</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>6 if signed (a sign and 5 digits) or 5 if unsigned (5 digits).</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>11 if signed (a sign and 10 digits) or 10 if unsigned (10 digits).</td>
</tr>
<tr>
<td>SQL_BIGINT</td>
<td>20 (a sign and 19 digits if signed or 20 digits if unsigned).</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>14 (a sign, 7 digits, a decimal point, the letter E, a sign, and 2 digits).</td>
</tr>
<tr>
<td>SQL_FLOAT, SQL_DOUBLE</td>
<td>24 (a sign, 15 digits, a decimal point, the letter E, a sign, and 3 digits).</td>
</tr>
<tr>
<td>All binary types[a]</td>
<td>The defined or maximum (for variable types) length of the column times 2. (Each binary byte is represented by a 2-digit hexadecimal number.)</td>
</tr>
<tr>
<td>SQL_TYPE_DATE</td>
<td>10 (a date in the format yyyy-mm-dd).</td>
</tr>
<tr>
<td>SQL_TYPE_TIME</td>
<td>8 (a time in the format hh:mm:ss) - or - 9 + s (a time in the format hh:mm:ss[.fff...], where s is the fractional seconds precision).</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>19 (for a timestamp in the yyyy-mm-dd hh:mm:ss format) - or - 20 + s (for a timestamp in the yyyy-mm-dd hh:mm:ss[.fff...] format, where s is the fractional seconds precision).</td>
</tr>
<tr>
<td>All interval data types</td>
<td>See Interval Data Type Length.</td>
</tr>
<tr>
<td>SQL_GUID</td>
<td>36 (the number of characters in the aaaaaaaaa-bbbbb-ccccccc-ddddd-eeeeeef format).</td>
</tr>
</tbody>
</table>

[a] If the driver cannot determine the column or parameter length of variable types, it returns SQL_NO_TOTAL.

Converting Data from SQL to C Data Types
When an application calls `SQLFetch`, `SQLFetchScroll`, or `SQLGetData`, the driver retrieves the data from the data source. If necessary, it converts the data from the data type in which the driver retrieved it to the data type specified by the `TargetType` argument in `SQLBindCol` or `SQLGetData`. Finally, it stores the data in the location pointed to by the `TargetValuePtr` argument in `SQLBindCol` or `SQLGetData` (and the SQL_DESC_DATA_PTR field of the ARD).

The following table shows the supported conversions from ODBC SQL data types to ODBC C data types. A filled circle indicates the default conversion for an SQL data type (the C data type to which the data will be converted when the value of `TargetType` is SQL_C_DEFAULT). A hollow circle indicates a supported conversion.

For an ODBC 3.x application working with an ODBC 2.x driver, conversion from driver-specific data types might not be supported.

The format of the converted data is not affected by the Windows® country setting.

The tables in the following sections describe how the driver or data source converts data retrieved from the data source; drivers are required to support conversions to all ODBC C data types from the ODBC SQL data types that they support. For a given ODBC SQL data type, the first column of the table lists the legal input values of the `TargetType` argument in `SQLBindCol` and `SQLGetData`. The second column lists the outcomes of a test, often using the `BufferLength` argument specified in `SQLBindCol` or `SQLGetData`, which the driver performs to determine whether it can convert the data. For each outcome, the third and fourth columns list the values placed in the buffers specified by the `TargetValuePtr` and `StrLen_or_IndPtr` arguments specified in `SQLBindCol` or `SQLGetData` after the driver has attempted to convert the data. (The `StrLen_or_IndPtr` argument corresponds to the SQL_DESC_OCTET_LENGTH_PTR field of the ARD.) The last column lists the SQLSTATE returned for each outcome by `SQLFetch`, `SQLFetchScroll`, or `SQLGetData`.

If the `TargetType` argument in `SQLBindCol` or `SQLGetData` contains an identifier for an ODBC C data type not shown in the table for a given ODBC SQL data type, `SQLFetch`, `SQLFetchScroll`, or `SQLGetData` returns SQLSTATE 07006 (Restricted data type attribute violation). If the `TargetType` argument contains an identifier that specifies a conversion from a driver-specific SQL data type to an ODBC C data type and this conversion is not supported by the driver, `SQLFetch`, `SQLFetchScroll`, or `SQLGetData` returns SQLSTATE HYC00 (Optional feature not implemented).

Although it is not shown in the tables, the driver returns SQL_NULL_DATA in the buffer specified by the `StrLen_or_IndPtr` argument when the SQL data value is NULL. For an explanation of the use of `StrLen_or_IndPtr` when multiple calls are made to retrieve data, see the `SQLGetData` function description. When SQL data is converted to character C data, the character count returned in `StrLen_or_IndPtr` does not include the null-termination byte. If `TargetValuePtr` is a null pointer, `SQLGetData` returns SQLSTATE HY009 (Invalid use of null pointer); in `SQLBindCol`, this unbinds the column.

The following terms and conventions are used in the tables:

- **Byte length of data** is the number of bytes of C data available to return in `*TargetValuePtr`, whether or not the data will be truncated before it is returned to the application. For string data, this does not include the space for the null-termination character.
- **Character byte length** is the total number of bytes needed to display the data in character format. This is as defined for each C data type in the section Display Size, except that character byte length is in bytes while the display size is in characters.
- Words in italics represent function arguments or elements of the SQL grammar. For the syntax of grammar elements, see Appendix C: SQL Grammar.

This section contains the following topics.

- SQL to C: Character
- SQL to C: Numeric
- SQL to C: Bit
- SQL to C: Binary
- SQL to C: Date
- SQL to C: GUID
- SQL to C: Time
- SQL to C: Timestamp
- SQL to C: Year-Month Intervals
- SQL to C: Day-Time Intervals
- SQL to C Data Conversion Examples
The following table shows the ODBC C data types to which character SQL data may be converted. For an explanation of the columns and terms in the table, see Converting Data from SQL to C Data Types.

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>Test</th>
<th>*TargetValuePtr</th>
<th>*StrLen_or_IndPtr</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_CHAR</td>
<td>Byte length of data &lt; BufferLength</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a 01004</td>
</tr>
<tr>
<td></td>
<td>Byte length of data &gt;= BufferLength</td>
<td>Truncated data</td>
<td>Length of data in bytes</td>
<td></td>
</tr>
<tr>
<td>SQL_C_WCHAR</td>
<td>Character length of data &lt; BufferLength</td>
<td>Data</td>
<td>Length of data in characters</td>
<td>n/a 01004</td>
</tr>
<tr>
<td></td>
<td>Character length of data &gt;= BufferLength</td>
<td>Truncated data</td>
<td>Length of data in characters</td>
<td></td>
</tr>
<tr>
<td>SQL_C_STINYINT SQL_C_UTINYINT SQL_C_TINYINT SQL_C_SBIGINT SQL_C_UBIGINT SQL_C_SSHORT SQL_C_USHORT SQL_C_SLONG SQL_C_ULONG SQL_C_LONG SQL_C_NUMERIC</td>
<td>Data converted without truncation[b]</td>
<td>Data</td>
<td>Number of bytes of the C data type</td>
<td>n/a 01507 22003 22018</td>
</tr>
<tr>
<td></td>
<td>Data converted with truncation of fractional digits[a]</td>
<td>Truncated data</td>
<td>Number of bytes of the C data type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conversion of data would result in loss of whole (as opposed to fractional) digits[a]</td>
<td>Undefined</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data is not a numeric-literal[b]</td>
<td>Undefined</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>SQL_C_FLOAT SQL_C_DOUBLE</td>
<td>Data is within the range of the data type to which the number is being converted[a]</td>
<td>Data</td>
<td>Size of the C data type</td>
<td>n/a 22003 22018</td>
</tr>
<tr>
<td></td>
<td>Data is outside the range of the data type to which the number is being converted[a]</td>
<td>Undefined</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data is not a numeric-literal[b]</td>
<td>Undefined</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>SQL_C_BIT</td>
<td>Data is 0 or 1</td>
<td>Data</td>
<td>1[b]</td>
<td>n/a 01507 22003 22018</td>
</tr>
<tr>
<td></td>
<td>Data is greater than 0, less than 2, and not equal to 1</td>
<td>Truncated data</td>
<td>1[b]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data is less than 0 or greater than or equal to 2</td>
<td>Undefined</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data is not a numeric-literal</td>
<td>Undefined</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>SQL_C_BINARY</td>
<td>Byte length of data &lt;= BufferLength</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a 01004</td>
</tr>
<tr>
<td></td>
<td>Byte length of data &gt; BufferLength</td>
<td>Truncated data</td>
<td>Length of data</td>
<td></td>
</tr>
<tr>
<td>SQL_C_TYPE_DATE</td>
<td>Data value is a valid date-value[a]</td>
<td>Data</td>
<td>6[b]</td>
<td>n/a 01507 22018</td>
</tr>
<tr>
<td></td>
<td>Data value is a valid timestamp-value; time portion is zero[a]</td>
<td>Data</td>
<td>6[b]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data value is a valid timestamp-value; time portion is nonzero[a],[c]</td>
<td>Truncated data</td>
<td>6[b]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data value is not a valid date-value or timestamp-value[a]</td>
<td>Undefined</td>
<td>Undefined</td>
<td></td>
</tr>
<tr>
<td>SQL_C_TYPE_TIME</td>
<td>Data value is a valid time-value and the fractional seconds value is 0[a]</td>
<td>Data</td>
<td>6[b]</td>
<td>n/a 01507 22018</td>
</tr>
<tr>
<td></td>
<td>Data value is a valid timestamp-value or a valid time-value; fractional seconds portion is zero[a],[d]</td>
<td>Data</td>
<td>6[b]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data value is a valid timestamp-value</td>
<td>Truncated data</td>
<td>6[b]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data value is not a valid timestamp-value</td>
<td>Undefined</td>
<td>Undefined</td>
<td></td>
</tr>
</tbody>
</table>
Data value is a valid timestamp-value; fractional seconds portion is nonzero[a], [d], [e]
Data value is not a valid time-value or timestamp-value[a]

SQL_C_TYPE_TIMESTAMP

Data value is a valid timestamp-value; fractional seconds portion not truncated[a]
Data value is a valid timestamp-value or a valid time-value; fractional seconds portion truncated[a]
Data value is a valid date-value[a]
Data value is a valid time-value[a]
Data value is not a valid date-value, time-value, or timestamp-value[a]

Data
Truncated data
Data[f]
Data[g]
Undefined

16[b]
16[b]
16[b]
16[b]
16[b]

Data
Truncated data
Undefined
Undefined

Length of data in bytes
Length of data in bytes
Undefined
Undefined

n/a
n/a
22018
22018

[a] The value of BufferLength is ignored for this conversion. The driver assumes that the size of *TargetValuePtr is the size of the C data type.
[b] This is the size of the corresponding C data type.
[c] The time portion of the timestamp-value is truncated.
[d] The date portion of the timestamp-value is ignored.
[e] The fractional seconds portion of the timestamp is truncated.
[f] The time fields of the timestamp structure are set to zero.
[g] The date fields of the timestamp structure are set to the current date.

When character SQL data is converted to numeric, date, time, timestamp, or interval C data, leading and trailing spaces are ignored.

SQL to C: Numeric

The identifiers for the numeric ODBC SQL data types are:

SQL_DECIMAL
SQL_BIGINT
SQL_NUMERIC
SQL_REAL
SQL_TINYINT
SQL_FLOAT
SQL_SMALLINT
SQL_DOUBLE SQL_INTEGER

The following table shows the ODBC C data types to which numeric SQL data may be converted. For an explanation of the columns and terms in the table, see
### Converting Data from SQL to C Data Types

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>Test</th>
<th>*TargetValuePtr</th>
<th>*StrLen_or_IndPtr</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_CHAR</td>
<td>Character byte length &lt; BufferLength</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Number of whole (as opposed to fractional) digits &lt; BufferLength</td>
<td>Truncated data</td>
<td>Undefined</td>
<td>01004</td>
</tr>
<tr>
<td></td>
<td>Number of whole (as opposed to fractional) digits &gt;= BufferLength</td>
<td>Undefined</td>
<td>Length of data in bytes</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_WCHAR</td>
<td>Character length &lt; BufferLength</td>
<td>Data</td>
<td>Length of data in characters</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Number of whole (as opposed to fractional) digits &lt; BufferLength</td>
<td>Truncated data</td>
<td>Undefined</td>
<td>01004</td>
</tr>
<tr>
<td></td>
<td>Number of whole (as opposed to fractional) digits &gt;= BufferLength</td>
<td>Undefined</td>
<td>Length of data in characters</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_STINYINT</td>
<td>Data converted without truncation[a]</td>
<td>Data</td>
<td>Size of the C data type</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_C_UTINYINT</td>
<td>Data converted with truncation of fractional digits[a]</td>
<td>Truncated data</td>
<td>Size of the C data type</td>
<td>01S07</td>
</tr>
<tr>
<td>SQL_C_TINYINT</td>
<td>Conversion of data would result in loss of whole (as opposed to fractional) digits[a]</td>
<td>Undefined</td>
<td>Undefined</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_SBIGINT</td>
<td>SQL_C_UBIGINT</td>
<td>SQL_C_SSHORT</td>
<td>SQL_C_USHORT</td>
<td>SQL_C_SHORT</td>
</tr>
<tr>
<td>SQL_C_FLOAT</td>
<td>Data is within the range of the data type to which the number is being converted[a]</td>
<td>Data</td>
<td>Size of the C data type</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_C_DOUBLE</td>
<td>Data is outside the range of the data type to which the number is being converted[a]</td>
<td>Undefined</td>
<td>Undefined</td>
<td>01004</td>
</tr>
<tr>
<td>SQL_C_BIT</td>
<td>Data is 0 or 1[a]</td>
<td>Data</td>
<td>1[b]</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Data is greater than 0, less than 2, and not equal to 1[a]</td>
<td>Truncated data</td>
<td>1[b]</td>
<td>01S07</td>
</tr>
<tr>
<td></td>
<td>Data is less than 0 or greater than or equal to 2[a]</td>
<td>Undefined</td>
<td>Undefined</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_BINARY</td>
<td>Byte length of data &lt;= BufferLength</td>
<td>Data</td>
<td>Length of data</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Byte length of data &gt; BufferLength</td>
<td>Undefined</td>
<td>Undefined</td>
<td>22003</td>
</tr>
<tr>
<td></td>
<td>Fractional seconds portion truncated</td>
<td>Truncated data</td>
<td>Undefined</td>
<td>01S07</td>
</tr>
<tr>
<td></td>
<td>Whole part of number</td>
<td>Undefined</td>
<td>Length of data in bytes</td>
<td>22015</td>
</tr>
</tbody>
</table>
The value of `BufferLength` is ignored for this conversion. The driver assumes that the size of `*TargetValuePtr` is the size of the C data type.

This is the size of the corresponding C data type.

This conversion is supported only for the exact numeric data types (SQL_DECIMAL, SQL_NUMERIC, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, and SQL_BIGINT). It is not supported for the approximate numeric data types (SQL_REAL, SQL_FLOAT, or SQL_DOUBLE).

### SQL_C_NUMERIC and SQLSetDescField

The **SQLSetDescField Function** is required to perform manual binding with SQL_C_NUMERIC values. (Note that SQLSetDescField was added in ODBC 3.0.) To perform manual binding, you must first get the descriptor handle.

```c
if (!fCType == SQL_C_NUMERIC) {
    // special processing required for NUMERIC to get right scale & precision
    // Modify the fields in the implicit application parameter descriptor
    SQLHDESC hdesc = NULL;

    // Use SQL_ATTR_APP_ROW_DESC for calls to SQLBindCol()
    // Use SQL_ATTR_APP_PARAM_DESC for calls to SQLBindParameter()
    // retcode = SQLGetStmtAttr(hstmt, SQL_ATTR_APP_ROW_DESC, &hdesc, 0, NULL);
    retcode = SQLGetStmtAttr(hstmt, SQL_ATTR_APP_PARAM_DESC, &hdesc, 0, NULL);
    if (!SQL_CALL_SUCCESS(retcode)) {
        printf("SQLGetStmtAttr failed\n");
        i = 1;
        sqlstate[7] = 'O';
        while (SQLGetDiagRec(SQL_HANDLE_STMT, hstmt, i, sqlstate, &NativeError, wrkbuf, sizeof(wrkbuf), &len) != SQL_NO_DATA) {
            printf("i\nCase\tPrecision\tScale\tNativeError\tsqlstate\twrkbuf\n");
            i = 1;
            sqlstate[7] = 'O';
            continue;
        }
    }
    retcode = SQLSetDescField(hdesc, iCol, SQL_DESC_TYPE, (SQLPOINTER) SQL_C_NUMERIC, 0);
    if (!SQL_CALL_SUCCESS(retcode))
        goto error;
    retcode = SQLSetDescField(hdesc, iCol, SQL_DESC_PRECISION, (SQLPOINTER) num.precision, 0);
    if (!SQL_CALL_SUCCESS(retcode))
        goto error;
    retcode = SQLSetDescField(hdesc, iCol, SQL_DESC_SCALE, (SQLPOINTER) num.scale, 0);
    if (!SQL_CALL_SUCCESS(retcode))
        goto error;
    retcode = SQLSetDescField(hdesc, iCol, SQL_DESC_DATA_PTR, (SQLPOINTER) &num, sizeof(num));
}
```

### SQL to C: Bit

The identifier for the bit ODBC SQL data type is:

SQL_BIT

The following table shows the ODBC C data types to which bit SQL data may be converted. For an explanation of the columns and terms in the table, see Converting Data from SQL to C Data Types.
### SQL to C: Binary

The identifiers for the binary ODBC SQL data types are:

- **SQL_BINARY**
- **SQL_VARBINARY**
- **SQL_LONGVARBINARY**

The following table shows the ODBC C data types to which binary SQL data may be converted. For an explanation of the columns and terms in the table, see [Converting Data from SQL to C Data Types](#).

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>Test</th>
<th>*TargetValuePtr</th>
<th>*StrLen_or_IndPtr</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_CHAR</td>
<td>(Byte length of data) * 2 &lt; BufferLength</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>(Byte length of data) * 2 &gt;= BufferLength</td>
<td>Truncated data</td>
<td>Length of data in bytes</td>
<td>01004</td>
</tr>
<tr>
<td>SQL_C_WCHAR</td>
<td>(Character length of data) * 2 &lt; BufferLength</td>
<td>Data</td>
<td>Length of data in characters</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>(Character length of data) * 2 &gt;= BufferLength</td>
<td>Truncated data</td>
<td>Length of data in characters</td>
<td>01004</td>
</tr>
<tr>
<td>SQL_C_BINARY</td>
<td>Byte length of data &lt;= BufferLength</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Byte length of data &gt; BufferLength</td>
<td>Truncated data</td>
<td>Length of data in bytes</td>
<td>01004</td>
</tr>
</tbody>
</table>

When binary SQL data is converted to character C data, each byte (8 bits) of source data is represented as two ASCII characters. These characters are the ASCII character representation of the number in its hexadecimal form. For example, a binary 00000001 is converted to "01" and a binary 11111111 is converted to "FF".

The driver always converts individual bytes to pairs of hexadecimal digits and terminates the character string with a null byte. Because of this, if BufferLength is even and is less than the length of the converted data, the last byte of the *TargetValuePtr buffer is not used. (The converted data requires an even number of bytes, the next-to-last byte is a null byte, and the last byte cannot be used.)

#### Note

Application developers are discouraged from binding binary SQL data to a character C data type. This conversion is usually inefficient and slow.
SQL to C: Date

The identifier for the date ODBC SQL data type is:

SQL_TYPE_DATE

The following table shows the ODBC C data types to which date SQL data may be converted. For an explanation of the columns and terms in the table, see Converting Data from SQL to C Data Types.

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>Test</th>
<th>*TargetValuePtr</th>
<th>*StrLen_or_IndPtr</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_CHAR</td>
<td>BufferLength &gt; Character byte length</td>
<td>Data</td>
<td>10</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>11 &lt;= BufferLength &lt;= Character byte length</td>
<td>Truncated data</td>
<td>Length of data in bytes</td>
<td>01004</td>
</tr>
<tr>
<td></td>
<td>BufferLength &lt; 11</td>
<td>Undefined</td>
<td>Undefined</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_WCHAR</td>
<td>BufferLength &gt; Character length</td>
<td>Data</td>
<td>10</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>11 &lt;= BufferLength &lt;= Character length</td>
<td>Truncated data</td>
<td>Length of data in characters</td>
<td>01004</td>
</tr>
<tr>
<td></td>
<td>BufferLength &lt; 11</td>
<td>Undefined</td>
<td>Undefined</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_BINARY</td>
<td>Byte length of data &lt;= BufferLength</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Byte length of data &gt; BufferLength</td>
<td>Undefined</td>
<td>Undefined</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_TYPE_DATE</td>
<td>None[a]</td>
<td>Data</td>
<td>6[c]</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_C_TYPE_TIMESTAMP</td>
<td>None[a]</td>
<td>Data[b]</td>
<td>16[c]</td>
<td>n/a</td>
</tr>
</tbody>
</table>

[a] The value of BufferLength is ignored for this conversion. The driver assumes that the size of *TargetValuePtr is the size of the C data type.
[b] The time fields of the timestamp structure are set to zero.
[c] This is the size of the corresponding C data type.

When date SQL data is converted to character C data, the resulting string is in the "yyyy-mm-dd" format. This format is not affected by the Windows® country setting.

© 2016 Microsoft

SQL to C: GUID

The identifier for the GUID ODBC SQL data type is:

SQL_GUID

The following table shows the ODBC C data types to which GUID SQL data may be converted. For an explanation of the columns and terms in the table, see Converting Data from SQL to C Data Types.

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>Test</th>
<th>*TargetValuePtr</th>
<th>*StrLen_or_IndPtr</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_CHAR</td>
<td>BufferLength &gt; Character byte length</td>
<td>Data</td>
<td>36</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>BufferLength &lt; 37</td>
<td>Undefined</td>
<td>Undefined</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_WCHAR</td>
<td>BufferLength &gt; Character length</td>
<td>Data</td>
<td>36</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>BufferLength &lt; 37</td>
<td>Undefined</td>
<td>Undefined</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_BINARY</td>
<td>Byte length of data &lt;= BufferLength</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Byte length of data &gt; BufferLength</td>
<td>Undefined</td>
<td>Undefined</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_GUID</td>
<td>None[a]</td>
<td>Data</td>
<td>16[b]</td>
<td>n/a</td>
</tr>
</tbody>
</table>

[a] The value of BufferLength is ignored for this conversion. The driver assumes that the size of *TargetValuePtr is the size of the C data type.
SQL to C: Time

The identifier for the time ODBC SQL data type is:

SQL_TYPE_TIME

The following table shows the ODBC C data types to which time SQL data may be converted. For an explanation of the columns and terms in the table, see Converting Data from SQL to C Data Types.

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>Test</th>
<th>*TargetValuePtr</th>
<th>*StrLen_or_IndPtr</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_CHAR</td>
<td>BufferLength &gt; Character byte length 9 &lt;= BufferLength &lt;= Character byte length BufferLength &lt; 9</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Truncated data[a] Length of data in bytes</td>
<td>01004</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_WCHAR</td>
<td>BufferLength &gt; Character length 9 &lt;= BufferLength &lt;= Character length BufferLength &lt; 9</td>
<td>Data</td>
<td>Length of data in characters</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Truncated data[a] Length of data in characters</td>
<td>01004</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_BINARY</td>
<td>Byte length of data &lt;= BufferLength Byte length of data &gt; BufferLength</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Undefined Length of data in bytes</td>
<td>22003</td>
<td></td>
</tr>
<tr>
<td>SQL_C_TYPE_TIME</td>
<td>None[b]</td>
<td>Data</td>
<td>6[d]</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_C_TYPE_TIMESTAMP</td>
<td>None[b]</td>
<td>Data[c]</td>
<td>16[d]</td>
<td>n/a</td>
</tr>
</tbody>
</table>

[a] The fractional seconds of the time are truncated.
[b] The value of BufferLength is ignored for this conversion. The driver assumes that the size of *TargetValuePtr is the size of the C data type.
[c] The date fields of the timestamp structure are set to the current date, and the fractional seconds field of the timestamp structure is set to zero.
[d] This is the size of the corresponding C data type.

When time SQL data is converted to character C data, the resulting string is in the "hh:mm:ss" format. This format is not affected by the Windows® country setting.

SQL to C: Timestamp

The identifier for the timestamp ODBC SQL data type is:

SQL_TYPE_TIMESTAMP

The following table shows the ODBC C data types to which timestamp SQL data can be converted. For an explanation of the columns and terms in the table, see Converting Data from SQL to C Data Types.

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>Test</th>
<th>*TargetValuePtr</th>
<th>*StrLen_or_IndPtr</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_CHAR</td>
<td>BufferLength &gt; Character byte length 20 &lt;= BufferLength &lt;= Character byte length BufferLength &lt; 20</td>
<td>Data</td>
<td>Length of data in bytes</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Truncated data[b] Length of data in bytes</td>
<td>01004</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_WCHAR</td>
<td>BufferLength &gt; Character length 20 &lt;= BufferLength &lt;= Character length BufferLength &lt; 20</td>
<td>Data</td>
<td>Length of data in characters</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Truncated data[b] Length of data in characters</td>
<td>01004</td>
<td>22003</td>
</tr>
</tbody>
</table>
When timestamp SQL data is converted to character C data, the resulting string is in the "yyyy-mm-dd hh:mm:ss[....]" format, where up to nine digits can be used for fractional seconds. This format is not affected by the Windows® country setting. (Except for the decimal point and fractional seconds, the entire format must be used, regardless of the precision of the timestamp SQL data type.)

### SQL to C: Year–Month Intervals

The identifiers for the year-month interval ODBC SQL data types are:

- SQL_INTERVAL_YEAR
- SQL_INTERVAL_MONTH
- SQL_INTERVAL_YEAR_TO_MONTH

The following table shows the ODBC C data types to which year-month interval SQL data may be converted. For an explanation of the columns and terms in the table, see Converting Data from SQL to C Data Types.
A year-month interval SQL type can be converted to any year-month interval C type.

If the interval precision is a single field (one of YEAR or MONTH), the interval SQL type can be converted to any exact numeric (SQL_C_STINYINT, SQL_C_UTINYINT, SQL_C_USHORT, SQL_C_SHORT, SQL_C_SLONG, SQL_C_ULONG, or SQL_C_NUMERIC).

The default conversion of an interval SQL type is to the corresponding C interval data type. The application then binds the column or parameter (or sets the SQL_DESC_DATA_PTR field in the appropriate record of the ARD) to point to the initialized SQL_INTERVAL_STRUCT structure (or passes a pointer to the SQL_INTERVAL_STRUCT structure as the TargetValuePtr argument in a call to SQLGetData).

© 2016 Microsoft

### SQL to C: Day–Time Intervals

The identifiers for the day-time interval ODBC SQL data types are:

- SQL_INTERVAL_DAY
- SQL_INTERVAL_DAY_TO_MINUTE
- SQL_INTERVAL_HOUR
- SQL_INTERVAL_DAY_TO_SECOND
- SQL_INTERVAL_MINUTE
- SQL_INTERVAL_HOUR_TO_MINUTE
- SQL_INTERVAL_SECOND
- SQL_INTERVAL_HOUR_TO_SECOND
- SQL_INTERVAL_DAY_TO_HOUR
- SQL_INTERVAL_MINUTE_TO_SECOND

The following table shows the ODBC C data types to which day-time interval SQL data may be converted. For an explanation of the columns and terms in the table, see Converting Data from SQL to C Data Types.

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>Test</th>
<th>*TargetValuePtr</th>
<th>*StrLen_or_IndPtr</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>All day-time C interval types</td>
<td></td>
<td>Data</td>
<td>Length of data</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_C_STINYINT[b] SQL_C_UTINYINT[b] SQL_C_USHORT[b] SQL_C_SHORT[b] SQL_C_SLONG[b] SQL_C_ULONG[b] SQL_C_NUMERIC[b] SQL_C_BIGINT[b]</td>
<td></td>
<td>Interval precision was a single field and the data was converted without truncation</td>
<td>Length of data</td>
<td>n/a</td>
</tr>
</tbody>
</table>
A day-time interval SQL type can be converted to any day-time interval C type.

If the interval precision is a single field (one of DAY, HOUR, MINUTE, or SECOND), the interval SQL type can be converted to any exact numeric (SQL_C_STINYINT, SQL_C_UTINYINT, SQL_C_USHORT, SQL_C_SHORT, SQL_C_SLONG, SQL_C_ULONG, or SQL_C_NUMERIC).

The default conversion of an interval SQL type is to the corresponding C interval data type. The application then binds the column or parameter (or sets the SQL_DESC_DATA_PTR field in the appropriate record of the ARD) to point to the initialized SQL_INTERVAL_STRUCT structure (or passes a pointer to the SQL_INTERVAL_STRUCT structure as the TargetValuePtr argument in a call to SQLGetData).

The following example demonstrates how to transfer data from a column of type SQL_INTERVAL_DAY_TO_MINUTE into the SQL_INTERVAL_STRUCT structure such that it comes back as a DAY_TO_HOUR interval.

```
SQL_INTERVAL_STRUCT is;
SQLINTEGER cbValue;
SQLUINTEGER days, hours;

// Execute a select statement; "interval_column" is a column
// whose data type is SQL_INTERVAL_DAY_TO_MINUTE.
SQLExecDirect(hstmt, "SELECT interval_column FROM table", SQL_NTS);

// Bind
SQLBindCol(hstmt, 1, SQL_C_INTERVAL_DAY_TO_MINUTE, &is, sizeof(SQL_INTERVAL_STRUCT), &cbValue);

// Fetch
SQLFetch(hstmt);

// Process data
days = is.intval.day_second.day;
hours = is.intval.day_second.hour;
```

SQL to C Data Conversion Examples

The examples shown in the following table illustrate how the driver converts SQL data to C data:

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>SQL data value</th>
<th>C type identifier</th>
<th>Buffer length</th>
<th>*TargetValuePtr</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>abcde</td>
<td>SQL_C_CHAR</td>
<td>7</td>
<td>abcde\0[a]</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_CHAR</td>
<td>abcde</td>
<td>SQL_C_CHAR</td>
<td>6</td>
<td>abcde\0[a]</td>
<td>01004</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>1234.56</td>
<td>SQL_C_CHAR</td>
<td>8</td>
<td>1234.56\0[a]</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>1234.56</td>
<td>SQL_C_CHAR</td>
<td>5</td>
<td>1234\0[a]</td>
<td>01004</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>1234.56</td>
<td>SQL_C_CHAR</td>
<td>4</td>
<td>----</td>
<td>22003</td>
</tr>
<tr>
<td>SQL Type</td>
<td>C Data Type</td>
<td>Parameter Type</td>
<td>Converted Value</td>
<td>SQL Value</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>----------------</td>
<td>------------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>SQL_C_FLOAT</td>
<td>ignored</td>
<td>1234.56</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>SQL_C_SSHORT</td>
<td>ignored</td>
<td>1234</td>
<td>01S07</td>
<td></td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>SQL_C_STINYINT</td>
<td>ignored</td>
<td>----</td>
<td>22003</td>
<td></td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>SQL_C_DOUBLE</td>
<td>ignored</td>
<td>1.2345678</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>SQL_C_FLOAT</td>
<td>ignored</td>
<td>1.234567</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>SQL_C_STINYINT</td>
<td>ignored</td>
<td>1</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>SQL_TYPE_DATE</td>
<td>SQL_C_CHAR</td>
<td>11</td>
<td>1992-12-31\0[a]</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>SQL_TYPE_DATE</td>
<td>SQL_C_CHAR</td>
<td>10</td>
<td>-----</td>
<td>22003</td>
<td></td>
</tr>
<tr>
<td>SQL_TYPE_DATE</td>
<td>SQL_C_TIMESTAMP</td>
<td>ignored</td>
<td>1992,12,31, 0,0,0,0[b]</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>SQL_C_CHAR</td>
<td>23</td>
<td>1992-12-31 23:45:55.12\0[a]</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>SQL_C_CHAR</td>
<td>22</td>
<td>1992-12-31 23:45:55.1\0[a]</td>
<td>01004</td>
<td></td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>SQL_C_CHAR</td>
<td>18</td>
<td>----</td>
<td>22003</td>
<td></td>
</tr>
</tbody>
</table>

[a] "\0" represents a null-termination byte. The driver always null-terminates SQL_C_CHAR data.

[b] The numbers in this list are the numbers stored in the fields of the TIMESTAMP_STRUCT structure.

© 2016 Microsoft

Converting Data from C to SQL Data Types

When an application calls SQLExecute or SQLExecDirect, the driver retrieves the data for any parameters bound with SQLBindParameter from storage locations in the application. When an application calls SQLSetPos, the driver retrieves the data for an update or add operation from columns bound with SQLBindCol. For data-at-execution parameters, the application sends the parameter data with SQLPutData. If necessary, the driver converts the data from the data type specified by the ValueType argument in SQLBindParameter to the data type specified by the ParameterType argument in SQLBindParameter, and then sends the data to the data source.

The following table shows the supported conversions from ODBC C data types to ODBC SQL data types. A filled circle indicates the default conversion for an SQL data type (the C data type from which the data will be converted when the value of ValueType or the SQL_DESC_CONCISE_TYPE descriptor field is SQL_C_DEFAULT). A hollow circle indicates a supported conversion.

The format of the converted data is not affected by the Windows® country setting.
The tables in the following sections describe how the driver or data source converts data sent to the data source; drivers are required to support conversions from all ODBC C data types to the ODBC SQL data types that they support. For a given ODBC C data type, the first column of the table lists the legal input values of the `ParameterType` argument in `SQLBindParameter`. The second column lists the outcomes of a test that the driver performs to determine if it can convert the data. The third column lists the SQLSTATE returned for each outcome by `SQLExecDirect`, `SQLExecute`, `SQLBulkOperations`, `SQLSetPos`, or `SQLPutData`. Data is sent to the data source only if SQL_SUCCESS is returned.

If the `ParameterType` argument in `SQLBindParameter` contains the identifier of an ODBC SQL data type that is not shown in the table for a given C data type, `SQLBindParameter` returns SQLSTATE 07006 (Restricted data type attribute violation). If the `ParameterType` argument contains a driver-specific identifier and the driver does not support the conversion from the specific ODBC C data type to that driver-specific SQL data type, `SQLBindParameter` returns SQLSTATE HYC00 (Optional feature not implemented).

If the `ParameterValuePtr` and `StrLen_or_IndPtr` arguments specified in `SQLBindParameter` are both null pointers, that function returns SQLSTATE HY009 (Invalid use of null pointer). Although it is not shown in the tables, an application sets the value of the length/indicator buffer pointed to by the `StrLen_or_IndPtr` argument of `SQLBindParameter` or the value of the `StrLen_or_IndPtr` argument of `SQLPutData` to SQL_NULL_DATA to specify a NULL SQL data value. (The `StrLen_or_IndPtr` argument corresponds to the SQL_DESC_OCTET_LENGTH_PTR field of the APD.) The application sets these values to SQL_NTS to specify that the value in `*ParameterValuePtr` in `SQLBindParameter` or `*DataPtr` in `SQLPutData` (pointed to by the SQL_DESC_DATA_PTR field of the APD) is a null-terminated string.

The following terms are used in the tables:

- **Byte length of data** — Number of bytes of SQL data available to send to the data source, whether or not the data will be truncated before it is sent to the data source. For string data, this does not include space for the null-termination character.
- **Column byte length** — Number of bytes required to store the data at the data source.
- **Character byte length** — Maximum number of bytes needed to display data in character form. This is as defined for each SQL data type in Display Size, except character byte length is in bytes, while the display size is in characters.
- **Number of digits** — Number of characters used to represent a number, including the minus sign, decimal point, and exponent (if needed).
- **Words in italics** — Elements of the SQL grammar. For the syntax of grammar elements, see Appendix C: SQL Grammar.

This section contains the following topics.

- C to SQL: Character
- C to SQL: Numeric
- C to SQL: Bit
The identifiers for the character ODBC C data type are:

- SQL_C_CHAR
- SQL_C_WCHAR

The following table shows the ODBC SQL data types to which C character data may be converted. For an explanation of the columns and terms in the table, see Converting Data from C to SQL Data Types.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>Byte length of data (\leq) Column length.</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>Byte length of data (&gt;) Column length.</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>Character length of data (\leq) Column length.</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>Character length of data (&gt;) Column length.</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>Data converted without truncation</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>Data converted with truncation of fractional digits[e]</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td>Conversion of data would result in loss of whole (as opposed to fractional) digits[e]</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>Data value is not a numeric-literal</td>
<td>22018</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>Data is within the range of the data type to which the number is being converted</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td>Data is outside the range of the data type to which the number is being converted</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>Data value is not a numeric-literal</td>
<td>22018</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>Data is 0 or 1</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Data is greater than 0, less than 2, and not equal to 1</td>
<td>22001</td>
</tr>
<tr>
<td></td>
<td>Data is less than 0 or greater than or equal to 2</td>
<td>22003</td>
</tr>
<tr>
<td></td>
<td>Data is not a numeric-literal</td>
<td>22018</td>
</tr>
<tr>
<td>SQL_BINARY</td>
<td>((\text{Byte length of data}) / 2 \leq \text{column byte length})</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_VARBINARY</td>
<td>((\text{Byte length of data}) / 2 &gt; \text{column byte length})</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARBINARY</td>
<td>Data value is not a hexadecimal value</td>
<td>22018</td>
</tr>
<tr>
<td>SQL_TYPE_DATE</td>
<td>Data value is a valid ODBC-date-literal</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Data value is a valid ODBC-timestamp-literal; time portion is zero
Data value is a valid ODBC-timestamp-literal; time portion is nonzero[a]
Data value is not a valid ODBC-date-literal or ODBC-timestamp-literal

SQL_TYPE_TIME
Data value is a valid ODBC-time-literal
Data value is a valid ODBC-timestamp-literal; fractional seconds portion is zero[b]
Data value is a valid ODBC-timestamp-literal; fractional seconds portion is nonzero[b]
Data value is not a valid ODBC-time-literal or ODBC-timestamp-literal

Data value is a valid ODBC-timestamp-literal; fractional seconds portion not truncated
Data value is a valid ODBC-timestamp-literal; fractional seconds portion truncated
Data value is a valid ODBC-date-literal[c]
Data value is a valid ODBC-time-literal[d]
Data value is not a valid ODBC-date-literal, ODBC-time-literal, or ODBC-timestamp-literal

All SQL interval types
Data value is a valid interval value; no truncation occurs
Data value is a valid interval value; the value in one of the fields is truncated
The data value is not a valid interval literal

[a] The time portion of the timestamp is truncated.
[b] The date portion of the timestamp is ignored.
[c] The time portion of the timestamp is set to zero.
[d] The date portion of the timestamp is set to the current date.
[e] The driver/data source effectively waits until the entire string has been received (even if the character data is sent in pieces by calls to SQLPutData) before attempting to perform the conversion.

When character C data is converted to numeric, date, time, or timestamp SQL data, leading and trailing blanks are ignored.

When character C data is converted to binary SQL data, each two bytes of character data are converted to a single byte (8 bits) of binary data. Each two bytes of character data represent a number in hexadecimal form. For example, “01” is converted to a binary 00000001 and “FF” is converted to a binary 11111111.

The driver always converts pairs of hexadecimal digits to individual bytes and ignores the null-termination byte. Because of this, if the length of the character string is odd, the last byte of the string (excluding the null-termination byte, if any) is not converted.

Note
Application developers are discouraged from binding character C data to a binary SQL data type. This conversion is usually inefficient and slow.

© 2016 Microsoft

C to SQL: Numeric

The identifiers for the numeric ODBC C data types are:

- SQL_C_STINYINT
- SQL_C_SLONG
- SQL_C_UTINYINT
- SQL_C_ULONG
- SQL_C_TINYINT
- SQL_C_LONG
- SQL_C_SSHORT
- SQL_C_FLOAT
- SQL_C_USHORT
- SQL_C_DOUBLE
- SQL_C_SHORT
The following table shows the ODBC SQL data types to which numeric C data may be converted. For an explanation of the columns and terms in the table, see Converting Data from C to SQL Data Types.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>Number of digits &lt;= Column byte length</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Number of digits &gt; Column byte length</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>Number of characters &lt;= Column character length</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Number of characters &gt; Column character length</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>Data converted without truncation or with truncated of fractional digits</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>Data converted with truncation of whole digits</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>Data is within the range of the data type to which the number is being converted</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td>Data is outside the range of the data type to which the number is being converted</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_DECIMAL[b]</td>
<td>Data is 0 or 1</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_NUMERIC[b]</td>
<td>Data is greater than 0, less than 2, and not equal to 1</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_TINYINT[b]</td>
<td>Data is less than 0 or greater than or equal to 2</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_SMALLINT[b]</td>
<td>Data not truncated.</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_INTEGER[b]</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_BIGINT[b]</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>Data not truncated.</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>Data not truncated.</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_INTERVAL_YEAR[a]</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_INTERVAL_MONTH[a]</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY[a]</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_INTERVAL_HOUR[a]</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_INTERVAL_MINUTE[a]</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_INTERVAL_SECOND[a]</td>
<td>Data truncated.</td>
<td>22015</td>
</tr>
</tbody>
</table>

[a] These conversions are supported only for the exact numeric data types (SQL_C_STINYINT, SQL_C_UTINYINT, SQL_C_SSHORT, SQL_C_USHORT, SQL_C_SLONG, SQL_C_ULONG, or SQL_C_NUMERIC). They are not supported for the approximate numeric data types (SQL_C_FLOAT or SQL_C_DOUBLE). Exact numeric C data types cannot be converted to an interval SQL type whose interval precision is not a single field.

[b] For the "n/a" case, a driver may optionally return SQL_SUCCESS_WITH_INFO and 01S07 when there is a fractional truncation.

The driver ignores the length/indicator value when converting data from the numeric C data types and assumes that the size of the data buffer is the size of the numeric C data type. The length/indicator value is passed in the StrLen_or_Ind argument in SQLPutData and in the buffer specified with the StrLen_or_IndPtr argument in SQLBindParameter. The data buffer is specified with the DataPtr argument in SQLPutData and the ParameterValuePtr argument in SQLBindParameter.

© 2016 Microsoft

C to SQL: Bit

The identifier for the bit ODBC C data type is:

SQL_C_BIT

The following table shows the ODBC SQL data types to which bit C data may be converted. For an explanation of the columns and terms in the table, see Converting Data from C to SQL Data Types.
### Converting Data from C to SQL Data Types

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>SQL_VARCHAR</td>
<td>None</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>SQL_WCHAR SQL_WVARCHAR</td>
<td>None</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>SQL_NUMERIC</td>
<td></td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td>SQL_SMALLINT</td>
<td></td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>SQL_BIGINT</td>
<td></td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>SQL_FLOAT</td>
<td></td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>SQL_TYPE_DATE</td>
<td></td>
</tr>
<tr>
<td>SQL_TYPE_TIME</td>
<td>SQL_TYPE_TIMESTAMP</td>
<td></td>
</tr>
</tbody>
</table>

The driver ignores the length/indicator value when converting data from the bit C data type and assumes that the size of the data buffer is the size of the bit C data type. The length/indicator value is passed in the `StrLen_or_Ind` argument in `SQLPutData` and in the buffer specified with the `StrLen_or_IndPtr` argument in `SQLBindParameter`. The data buffer is specified with the `DataPtr` argument in `SQLPutData` and the `ParameterValuePtr` argument in `SQLBindParameter`.

© 2016 Microsoft

## C to SQL: Binary

The identifier for the binary ODBC C data type is:

**SQL_C_BINARY**

The following table shows the ODBC SQL data types to which binary C data may be converted. For an explanation of the columns and terms in the table, see *Converting Data from C to SQL Data Types*.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>Byte length of data &lt;= Column byte length</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>Byte length of data &gt; Column byte length</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>Character length of data &lt;= Column character length</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>Character length of data &gt; Column character length</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>Byte length of data = SQL data length</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>Byte length of data &lt;&gt; SQL data length</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>SQL_TYPE_DATE</td>
<td></td>
</tr>
<tr>
<td>SQL_TYPE_TIME</td>
<td>SQL_TYPE_TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C to SQL: Date

The identifier for the date ODBC C data type is:

SQL_C_TYPE_DATE

The following table shows the ODBC SQL data types to which date C data may be converted. For an explanation of the columns and terms in the table, see Converting Data from C to SQL Data Types.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>Column byte length &gt;= 10</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>Column byte length &lt; 10</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>Data value is not a valid date</td>
<td>22008</td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>Column character length &gt;= 10</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>Column character length &lt; 10</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td>Data value is not a valid date</td>
<td>22008</td>
</tr>
<tr>
<td>SQL_TYPE_DATE</td>
<td>Data value is a valid date</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>Data value is a valid date[a]</td>
<td>22007</td>
</tr>
<tr>
<td></td>
<td>Data value is not a valid date</td>
<td></td>
</tr>
</tbody>
</table>

[a] The time portion of the timestamp is set to zero.

For information about what values are valid in a SQL_C_TYPE_DATE structure, see C Data Types, earlier in this appendix.

When date C data is converted to character SQL data, the resulting character data is in the "yyyy-mm-dd" format.

The driver ignores the length/indicator value when converting data from the date C data type and assumes that the size of the data buffer is the size of the date C data type. The length/indicator value is passed in the StrLen_or_Ind argument in SQLPutData and in the buffer specified with the StrLen_or_IndPtr argument in SQLBindParameter. The data buffer is specified with the DataPtr argument in SQLPutData and the ParameterValuePtr argument in SQLBindParameter.

C to SQL: GUID

The identifier for the GUID ODBC C data type is:

SQL_C_GUID

The following table shows the ODBC SQL data types to which GUID C data may be converted. For an explanation of the columns and terms in the table, see Converting Data from C to SQL Data Types.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>Column byte length &gt;= 36</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>Column byte length &lt; 36</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>Data value is not a valid GUID</td>
<td>22018</td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>Column character length &gt;= 36</td>
<td>n/a</td>
</tr>
</tbody>
</table>
The driver ignores the length/indicator value when converting data from the GUID C data type and assumes that the size of the data buffer is the size of the GUID C data type. The length/indicator value is passed in the `StrLen_or_Ind` argument in `SQLPutData` and in the buffer specified with the `StrLen_or_IndPtr` argument in `SQLBindParameter`. The data buffer is specified with the `DataPtr` argument in `SQLPutData` and the `ParameterValuePtr` argument in `SQLBindParameter`.

© 2016 Microsoft

C to SQL: Time

The identifier for the time ODBC C data type is:

```
SQL_C_TYPE_TIME
```

The following table shows the ODBC SQL data types to which time C data may be converted. For an explanation of the columns and terms in the table, see Converting Data from C to SQL Data Types.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>Column byte length &gt;= 8</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>Column byte length &lt; 8</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>Data value is not a valid time</td>
<td>22008</td>
</tr>
<tr>
<td>SQL_WCHAR</td>
<td>Column character length &gt;= 8</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>Column character length &lt; 8</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td>Data value is not a valid time</td>
<td>22008</td>
</tr>
<tr>
<td>SQL_TYPE_TIME</td>
<td>Data value is a valid time</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Data value is not a valid time</td>
<td>22007</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>Data value is a valid time[a]</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Data value is not a valid time</td>
<td>22007</td>
</tr>
</tbody>
</table>

[a] The date portion of the timestamp is set to the current date, and the fractional seconds portion of the timestamp is set to zero.

For information about what values are valid in a `SQL_C_TYPE_TIME` structure, see C Data Types, earlier in this appendix.

When time C data is converted to character SQL data, the resulting character data is in the "hh:mm:ss" format.

The driver ignores the length/indicator value when converting data from the time C data type and assumes that the size of the data buffer is the size of the time C data type. The length/indicator value is passed in the `StrLen_or_Ind` argument in `SQLPutData` and in the buffer specified with the `StrLen_or_IndPtr` argument in `SQLBindParameter`. The data buffer is specified with the `DataPtr` argument in `SQLPutData` and the `ParameterValuePtr` argument in `SQLBindParameter`.

© 2016 Microsoft

C to SQL: Timestamp

The identifier for the timestamp ODBC C data type is:

```
SQL_C_TYPE_TIMESTAMP
```

The following table shows the ODBC SQL data types to which timestamp C data may be converted. For an explanation of the columns and terms in the table, see Converting Data from C to SQL Data Types.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR</td>
<td>Column byte length &gt;= Character byte length</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL type identifier</td>
<td>Test</td>
<td>SQLSTATE</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>SQL_CHAR[a]</td>
<td>Column byte length &gt;= Character byte length</td>
<td>n/a</td>
</tr>
<tr>
<td>SQLVARCHAR[a]</td>
<td>Column byte length &lt; Character byte length[a]</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>Data value is not a valid interval literal</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_WCHAR[a]</td>
<td>Column character length &gt;= Character length of data</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>Column character length &lt; Character length of data[a]</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td>Data value is not a valid interval literal</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_TYPE_DATE</td>
<td>Time fields are zero</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Time fields are nonzero</td>
<td>22008</td>
</tr>
<tr>
<td></td>
<td>Data value does not contain a valid date</td>
<td>22007</td>
</tr>
<tr>
<td>SQL_TYPE_TIME</td>
<td>Fractional seconds fields are zero[a]</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Fractional seconds fields are nonzero[a]</td>
<td>22008</td>
</tr>
<tr>
<td></td>
<td>Data value does not contain a valid time</td>
<td>22007</td>
</tr>
<tr>
<td>SQL_TYPE_TIMESTAMP</td>
<td>Fractional seconds fields are not truncated</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>Fractional seconds fields are truncated</td>
<td>22008</td>
</tr>
<tr>
<td></td>
<td>Data value is not a valid timestamp</td>
<td>22007</td>
</tr>
</tbody>
</table>

[a] The date fields of the timestamp structure are ignored.

For information about what values are valid in a SQL_C_TIMESTAMP structure, see C Data Types, earlier in this appendix.

When timestamp C data is converted to character SQL data, the resulting character data is in the "yyyy-mm-dd hh:mm:ss[.f...]
" format.

The driver ignores the length/indicator value when converting data from the timestamp C data type and assumes that the size of the data buffer is the size of the timestamp C data type. The length/indicator value is passed in the StrLen_or_Ind argument in SQLPutData and in the buffer specified with the StrLen_or_IndPtr argument in SQLBindParameter. The data buffer is specified with the DataPtr argument in SQLPutData and the ParameterValuePtr argument in SQLBindParameter.

© 2016 Microsoft

C to SQL: Year–Month Intervals

The identifiers for the year-month interval ODBC C data types are:

```
SQL_C_INTERVAL_MONTH SQL_C_INTERVAL_YEAR SQL_C_INTERVAL_YEAR_TO_MONTH
```

The following table shows the ODBC SQL data types to which year-month interval C data may be converted. For an explanation of the columns and terms in the table, see Converting Data from C to SQL Data Types.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR[a]</td>
<td>Column byte length &gt;= Character byte length</td>
<td>n/a</td>
</tr>
<tr>
<td>SQLVARCHAR[a]</td>
<td>Column byte length &lt; Character byte length[a]</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>Data value is not a valid interval literal</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_WCHAR[a]</td>
<td>Column character length &gt;= Character length of data</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_WVARCHAR</td>
<td>Column character length &lt; Character length of data[a]</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR</td>
<td>Data value is not a valid interval literal</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_TINYINT[b]</td>
<td>Conversion of a single-field interval did not result in truncation of whole digits</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_SMALLINT[b]</td>
<td>Conversion resulted in truncation of whole digits</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_INTEGER[b]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_BIGINT[b]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_NUMERIC[b]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_DECIMAL[b]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C to SQL: Day–Time Intervals

The identifiers for the day-time interval ODBC C data types are:

- SQL_C_INTERVAL_DAY
- SQL_C_INTERVAL_HOUR
- SQL_C_INTERVAL_MINUTE
- SQL_C_INTERVAL_SECOND
- SQL_C_INTERVAL_DAY_TO_HOUR
- SQL_C_INTERVAL_DAY_TO_MINUTE
- SQL_C_INTERVAL_DAY_TO_SECOND
- SQL_C_INTERVAL_HOUR_TO_MINUTE
- SQL_C_INTERVAL_HOUR_TO_SECOND
- SQL_C_INTERVAL_MINUTE_TO_SECOND

The following table shows the ODBC SQL data types to which interval C data may be converted. For an explanation of the columns and terms in the table, see Converting Data from C to SQL Data Types.

<table>
<thead>
<tr>
<th>SQL type identifier</th>
<th>Test</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CHAR[a]</td>
<td>Column byte length &gt;= Character byte length</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_VARCHAR[a]</td>
<td>Column byte length &lt; Character byte length[a]</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR[a]</td>
<td>Data value is not a valid interval literal</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_WCHAR[a]</td>
<td>Column character length &gt;= Character length of data</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_WVARCHAR[a]</td>
<td>Column character length &lt; Character length of data[a]</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_WLONGVARCHAR[a]</td>
<td>Data value is not a valid interval literal</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_TINYINT[b]</td>
<td>Conversion of a single-field interval did not result in truncation of whole digits</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_SMALLINT[b] SQL_INTEGER[b]</td>
<td>Conversion resulted in truncation of whole digits</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_BIGINT[b] SQL_NUMERIC[b]</td>
<td>Conversion resulted in truncation of whole digits</td>
<td></td>
</tr>
<tr>
<td>SQL_DECIMAL[b]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY</td>
<td>Data value was converted without truncation of any fields</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_INTERVAL_HOUR</td>
<td>One or more fields of data value were truncated during conversion</td>
<td>22015</td>
</tr>
<tr>
<td>SQL_INTERVAL_MINUTE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_INTERVAL_SECOND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL_INTERVAL_DAY_TO_HOUR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SQL_INTERVAL_DAY_TO_MINUTE
SQL_INTERVAL_DAY_TO_SECOND
SQL_INTERVAL_HOUR_TO_MINUTE
SQL_INTERVAL_HOUR_TO_SECOND
SQL_INTERVAL_MINUTE_TO_SECOND

[a] All C interval data types can be converted to a character data type.

[b] If the type field in the interval structure is such that the interval is a single field (SQL_DAY, SQL_HOUR, SQL_MINUTE, or SQL_SECOND), the interval C type can be converted to any exact numeric (SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_DECIMAL, or SQL_NUMERIC).

The default conversion of an interval C type is to the corresponding day-time interval SQL type. The driver ignores the length/indicator value when converting data from the interval C data type and assumes that the size of the data buffer is the size of the interval C data type. The length/indicator value is passed in the StrLen_or_Ind argument in SQLPutData and in the buffer specified with the StrLen_or_IndPtr argument in SQLBindParameter. The data buffer is specified with the DataPtr argument in SQLPutData and the ParameterValuePtr argument in SQLBindParameter.

The following example demonstrates how to send interval C data stored in the SQL_INTERVAL_STRUCT structure into a database column. The interval structure contains a DAY_TO_SECOND interval; it will be stored in a database column of type SQL_INTERVAL_DAY_TO_MINUTE.

```c
SQL_INTERVAL_STRUCT is;
SQLINTEGER cbValue;

// Initialize the interval struct to contain the DAY_TO_SECOND
// interval "154 days, 22 hours, 44 minutes, and 10 seconds"
is.intval.day_second.day = 154;
is.intval.day_second.hour = 22;
is.intval.day_second.minute = 44;
is.intval.day_second.second = 10;
is.interval_sign = SQL_FALSE;

// Bind the dynamic parameter
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTERVAL_DAY_TO_SECOND, SQL_INTERVAL_DAY_TO_MINUTE, 0, 0, &is, sizeof(SQL_INTERVAL_STRUCT), &cbValue);

// Execute an insert statement; "interval_column" is a column
// whose data type is SQL_INTERVAL_DAY_TO_MINUTE.
SQLExecDirect(hstmt, "INSERT INTO table(interval_column) VALUES (?)", SQL_NTS);
```

C to SQL Data Conversion Examples

The following examples illustrate how the driver converts C data to SQL data:

<table>
<thead>
<tr>
<th>C type identifier</th>
<th>C data value</th>
<th>SQL type identifier</th>
<th>Column length</th>
<th>SQL data value</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_C_CHAR</td>
<td>abcdef0</td>
<td>SQL_CHAR</td>
<td>6</td>
<td>abcdef</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_C_CHAR</td>
<td>abcdef0</td>
<td>SQL_CHAR</td>
<td>5</td>
<td>abcd</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_C_CHAR</td>
<td>1234.56</td>
<td>SQL_DECIMAL</td>
<td>8(b)</td>
<td>1234.56</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_C_CHAR</td>
<td>1234.56</td>
<td>SQL_DECIMAL</td>
<td>7(b)</td>
<td>1234.5</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_C_CHAR</td>
<td>1234.56</td>
<td>SQL_DECIMAL</td>
<td>4</td>
<td>----</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_FLOAT</td>
<td>1234.56</td>
<td>SQL_FLOAT</td>
<td>n/a</td>
<td>1234.56</td>
<td>n/a</td>
</tr>
<tr>
<td>SQL_C_FLOAT</td>
<td>1234.56</td>
<td>SQL_INTEGER</td>
<td>n/a</td>
<td>1234</td>
<td>22001</td>
</tr>
<tr>
<td>SQL_C_FLOAT</td>
<td>1234.56</td>
<td>SQL_TINYINT</td>
<td>n/a</td>
<td>----</td>
<td>22003</td>
</tr>
<tr>
<td>SQL_C_TYPE_DATE</td>
<td>1992,12,31</td>
<td>SQL_CHAR</td>
<td>10</td>
<td>1992-12-31</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Appendix E: Scalar Functions

ODBC specifies the following types of scalar functions, with detailed information about each of these function types provided in the corresponding sections of this appendix. The function descriptions include associated syntax.

This appendix contains the following topics.

- String Functions
- Numeric Functions
- Time, Date, and Interval Functions
- System Functions
- Explicit Data Type Conversion Function
- SQL-92 CAST Function

ODBC does not mandate a data type for return values from scalar functions because the functions are often data source–specific. Applications should use the CONVERT scalar function whenever possible to force data type conversion.

ODBC and SQL–92 Scalar Functions

The tables in this appendix include functions that have been added in ODBC 3.0 to align with SQL-92. Those functions added for a particular type of scalar function, as defined in ODBC, are indicated in each section.

ODBC and SQL-92 classify their scalar functions differently. ODBC classifies scalar functions by argument type; SQL-92 classifies them by return value. For example, the EXTRACT function is classified as a timedate function by ODBC, because the extract-field argument is a datetime keyword and the extract-source argument is a datetime or interval expression. SQL-92, on the other hand, classifies EXTRACT as a numeric scalar function, because the return value is a numeric.

An application can determine which scalar functions a driver supports by calling SQLGetInfo. Information types are included both for ODBC and for SQL-92 classifications of scalar functions. Because these classifications are different, the support for some scalar functions may be indicated in information types that do not correspond to ODBC and SQL-92. For example, support for EXTRACT in ODBC is indicated by the SQL_TIMEDATE_FUNCTIONS information type; support for EXTRACT in SQL-92, on the other hand, is indicated by the SQL_SQL92_NUMERIC_VALUE_FUNCTIONS information type.

String Functions

The following table lists string manipulation functions. An application can determine which string functions are supported by a driver by calling SQLGetInfo with an information type of SQL_STRING_FUNCTIONS.

Remarks

Arguments denoted as string_exp can be the name of a column, a character-string-literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQLVARCHAR, or SQLLONGVARCHAR.

Arguments denoted as character_exp are a variable-length character string.
Arguments denoted as start, length, or count can be a numeric-literal or the result of another scalar function, where the underlying data type can be represented as SQL_TINYINT, SQL_SMALLINT, or SQL_INTEGER.

The string functions listed here are 1-based; that is, the first character in the string is character 1.

The BIT_LENGTH, CHAR_LENGTH, CHARACTER_LENGTH, OCTET_LENGTH, and POSITION string scalar functions have been added in ODBC 3.0 to align with SQL-92.

### Function Descriptions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII(string_exp) (ODBC 1.0)</td>
<td>Returns the ASCII code value of the leftmost character of string_exp as an integer.</td>
</tr>
<tr>
<td>BIT_LENGTH(string_exp) (ODBC 3.0)</td>
<td>Returns the length in bits of the string expression. Does not work only for string data types, therefore will not implicitly convert string_exp to string but instead will return the (internal) size of whatever datatype it is given.</td>
</tr>
<tr>
<td>CHAR(code) (ODBC 1.0)</td>
<td>Returns the character that has the ASCII code value specified by code. The value of code should be between 0 and 255; otherwise, the return value is data source-dependent.</td>
</tr>
<tr>
<td>CHAR_LENGTH(string_exp) (ODBC 3.0)</td>
<td>Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHARACTER_LENGTH function.)</td>
</tr>
<tr>
<td>CHARACTER_LENGTH(string_exp) (ODBC 3.0)</td>
<td>Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHARACTER_LENGTH function.)</td>
</tr>
<tr>
<td>CONCAT(string_exp1,string_exp2) (ODBC 1.0)</td>
<td>Returns a character string that is the result of concatenating string_exp2 to string_exp1. The resulting string is DBMS-dependent. For example, if the column represented by string_exp1 contained a NULL value, DB2 would return NULL but SQL Server would return the non-NULL string.</td>
</tr>
<tr>
<td>DIFFERENCE(string_exp1,string_exp2) (ODBC 2.0)</td>
<td>Returns an integer value that indicates the difference between the values returned by the SOUNDEX function for string_exp1 and string_exp2.</td>
</tr>
<tr>
<td>INSERT(string_exp1, start, length, string_exp2) (ODBC 1.0)</td>
<td>Returns a character string where length characters have been deleted from string_exp1, beginning at start, and where string_exp2 has been inserted into string_exp, beginning at start.</td>
</tr>
<tr>
<td>LCASE(string_exp) (ODBC 1.0)</td>
<td>Returns a string equal to that in string_exp, with all uppercase characters converted to lowercase.</td>
</tr>
<tr>
<td>LEFT(string_exp, count) (ODBC 1.0)</td>
<td>Returns the leftmost count characters of string_exp.</td>
</tr>
<tr>
<td>LENGTH(string_exp) (ODBC 1.0)</td>
<td>Returns the number of characters in string_exp, excluding trailing blanks. LENGTH only accepts strings. Therefore will implicitly convert string_exp to a string, and return the length of this string (not the internal size of the datatype).</td>
</tr>
<tr>
<td>LOCATE(string_exp1, string_exp2[, start]) (ODBC 1.0)</td>
<td>Returns the starting position of the first occurrence of string_exp1 within string_exp2. The search for the first occurrence of string_exp1 begins with the first character position in string_exp2 unless the optional argument, start, is specified. If start is specified, the search begins with the character position indicated by the value of start. The first character position in string_exp2 is indicated by the value 1. If string_exp1 is not found within string_exp2, the value 0 is returned.</td>
</tr>
<tr>
<td>OCTET_LENGTH(string_exp) (ODBC 3.0)</td>
<td>Returns the length in bytes of the string expression. The result is the smallest integer not less than the number of bits divided by 8. Does not work only for string data types, therefore will not implicitly convert string_exp to string but instead will return the (internal) size of whatever datatype it is given.</td>
</tr>
<tr>
<td>POSITION(character_exp IN character_exp) (ODBC 3.0)</td>
<td>Returns the position of the first character expression in the second character expression. The result is an exact numeric with an implementation-defined precision and a scale of 0.</td>
</tr>
<tr>
<td>REPEAT(string_exp, count) (ODBC 1.0)</td>
<td>Returns a character string composed of string_exp repeated count times.</td>
</tr>
<tr>
<td>REPLACE(string_exp1, string_exp2, string_exp3) (ODBC 1.0)</td>
<td>Search string_exp1 for occurrences of string_exp2, and replace with string_exp3.</td>
</tr>
<tr>
<td>RIGHT(string_exp, count) (ODBC 1.0)</td>
<td>Returns the rightmost count characters of string_exp.</td>
</tr>
</tbody>
</table>
RTRIM(string_exp) (ODBC 1.0)  | Returns the characters of string_exp with trailing blanks removed.
SOUNDEX(string_exp) (ODBC 2.0)  | Returns a data source-dependent character string representing the sound of the words in string_exp. For example, SQL Server returns a 4-digit SOUNDEX code; Oracle returns a phonetic representation of each word.
SPACE(count) (ODBC 2.0)  | Returns a character string consisting of count spaces.
SUBSTRING(string_exp, start, length) (ODBC 1.0)  | Returns a character string that is derived from string_exp, beginning at the character position specified by start for length characters.
UCASE(string_exp) (ODBC 1.0)  | Returns a string equal to that in string_exp, with all lowercase characters converted to uppercase.

**Numeric Functions**

The following table describes numeric functions that are included in the ODBC scalar function set. By calling SQLGetInfo with an information type of SQL_NUMERIC_FUNCTIONS, an application can determine which numeric functions are supported by a driver.

All numeric functions return values of data type SQL_FLOAT except for ABS, ROUND, TRUNCATE, SIGN, FLOOR, and CEILING, which return values of the same data type as the input parameters.

Arguments denoted as numeric_exp can be the name of a column, the result of another scalar function, or a numeric-literal, where the underlying data type could be represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, or SQL_DOUBLE.

Arguments denoted as float_exp can be the name of a column, the result of another scalar function, or a numeric-literal, where the underlying data type can be represented as SQL_FLOAT.

Arguments denoted as integer_exp can be the name of a column, the result of another scalar function, or a numeric-literal, where the underlying data type can be represented as SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, or SQL_BIGINT.

The CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP scalar functions have been added in ODBC 3.0 to align with SQL-92.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(numeric_exp) (ODBC 1.0)</td>
<td>Returns the absolute value of numeric_exp.</td>
</tr>
<tr>
<td>ACOS(float_exp) (ODBC 1.0)</td>
<td>Returns the arccosine of float_exp as an angle, expressed in radians.</td>
</tr>
<tr>
<td>ASIN(float_exp) (ODBC 1.0)</td>
<td>Returns the arcsine of float_exp as an angle, expressed in radians.</td>
</tr>
<tr>
<td>ATAN(float_exp) (ODBC 1.0)</td>
<td>Returns the arctangent of float_exp as an angle, expressed in radians.</td>
</tr>
<tr>
<td>ATAN2(float_exp1, float_exp2) (ODBC 2.0)</td>
<td>Returns the arctangent of the x and y coordinates, specified by float_exp1 and float_exp2, respectively, as an angle, expressed in radians.</td>
</tr>
<tr>
<td>CEILING(numeric_exp) (ODBC 1.0)</td>
<td>Returns the smallest integer greater than or equal to numeric_exp. The return value is of the same data type as the input parameter.</td>
</tr>
<tr>
<td>COS(float_exp) (ODBC 1.0)</td>
<td>Returns the cosine of float_exp, where float_exp is an angle expressed in radians.</td>
</tr>
<tr>
<td>COT(float_exp) (ODBC 1.0)</td>
<td>Returns the cotangent of float_exp, where float_exp is an angle expressed in radians.</td>
</tr>
<tr>
<td>DEGREES(numeric_exp) (ODBC 2.0)</td>
<td>Returns the number of degrees converted from numeric_exp radians.</td>
</tr>
<tr>
<td>EXP(float_exp) (ODBC 1.0)</td>
<td>Returns the exponential value of float_exp.</td>
</tr>
<tr>
<td>FLOOR(numeric_exp) (ODBC 1.0)</td>
<td>Returns the largest integer less than or equal to numeric_exp. The return value is of the same data type as the input parameter.</td>
</tr>
<tr>
<td>LOG(float_exp) (ODBC 1.0)</td>
<td>Returns the natural logarithm of float_exp.</td>
</tr>
<tr>
<td>LOG10(float_exp) (ODBC 2.0)</td>
<td>Returns the base 10 logarithm of float_exp.</td>
</tr>
<tr>
<td>MOD(integer_exp1, integer_exp2) (ODBC 1.0)</td>
<td>Returns the remainder (modulus) of integer_exp1 divided by integer_exp2.</td>
</tr>
<tr>
<td>PI() (ODBC 1.0)</td>
<td>Returns the constant value of pi as a floating-point value.</td>
</tr>
</tbody>
</table>
Time, Date, and Interval Functions

The following table lists time and date functions that are included in the ODBC scalar function set. An application can determine which time and date functions are supported by a driver by calling SQLGetInfo with an information type of SQL_TIMEDATE_FUNCTIONS.

Arguments denoted as timestamp_exp can be the name of a column, the result of another scalar function, or an ODBC-time-escape, ODBC-date-escape, or ODBC-timestamp-escape, where the underlying data type could be represented as SQL_CHAR, SQLVARCHAR, SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP.

Arguments denoted as date_exp can be the name of a column, the result of another scalar function, or an ODBC-date-escape or ODBC-timestamp-escape, where the underlying data type could be represented as SQL_CHAR, SQLVARCHAR, SQL_TYPE_DATE, or SQL_TYPE_TIMESTAMP.

Arguments denoted as time_exp can be the name of a column, the result of another scalar function, or an ODBC-time-escape or ODBC-timestamp-escape, where the underlying data type could be represented as SQL_CHAR, SQLVARCHAR, SQL_TYPE_DATE, or SQL_TYPE_TIMESTAMP.

The CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP timedate scalar functions have been added in ODBC 3.0 to align with SQL-92.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_DATE( ) (ODBC 3.0)</td>
<td>Returns the current date.</td>
</tr>
<tr>
<td>CURRENT_TIME[time-precision] (ODBC 3.0)</td>
<td>Returns the current local time. The time-precision argument determines the seconds precision of the returned value.</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP [timestamp-precision] (ODBC 3.0)</td>
<td>Returns the current local date and local time as a timestamp value. The timestamp-precision argument determines the seconds precision of the returned timestamp.</td>
</tr>
<tr>
<td>CURDATE( ) (ODBC 1.0)</td>
<td>Returns the current date.</td>
</tr>
<tr>
<td>CURTIME( ) (ODBC 1.0)</td>
<td>Returns the current local time.</td>
</tr>
<tr>
<td>DAYNAME(date_exp) (ODBC 2.0)</td>
<td>Returns a character string containing the data source-specific name of the day (for example, Sunday through Saturday or Sun. through Sat. for a data source that uses English, or Sonntag through Samstag for a data source that uses German) for the day portion of date_exp.</td>
</tr>
<tr>
<td>DAYOFMONTH(date_exp) (ODBC 1.0)</td>
<td>Returns the day of the month based on the month field in date_exp as an integer value in the range of 1–31.</td>
</tr>
<tr>
<td>DAYOFWEEK(date_exp) (ODBC 1.0)</td>
<td>Returns the day of the week based on the week field in date_exp as an integer value in the range of 1–7, where 1 represents Sunday.</td>
</tr>
<tr>
<td>DAYOFYEAR(date_exp) (ODBC 1.0)</td>
<td>Returns the day of the year based on the year field in date_exp as an integer value in the range of 1–366.</td>
</tr>
<tr>
<td>EXTRACT(extract-field FROM extract-source) (ODBC 3.0)</td>
<td>Returns the extract-field portion of the extract-source. The extract-source argument is a datetime or interval expression. The extract-field argument can be one of the following keywords: \n\nYEAR MONTH DAY HOUR MINUTE SECOND \n\nThe precision of the returned value is implementation-defined. The scale is 0 unless SECOND is specified, in which case the scale is determined by the time-precision of the argument.</td>
</tr>
</tbody>
</table>
HOUR(time_exp) (ODBC 1.0)  
Returns the hour based on the hour field in time_exp as an integer value in the range of 0–23.

MINUTE(time_exp) (ODBC 1.0)  
Returns the minute based on the minute field in time_exp as an integer value in the range of 0–59.

MONTH(date_exp) (ODBC 1.0)  
Returns the month based on the month field in date_exp as an integer value in the range of 1–12.

MONTHNAME(date_exp) (ODBC 2.0)  
Returns a character string containing the data source–specific name of the month (for example, January through December or Jan. through Dec. for a data source that uses English, or Januar through Dezember for a data source that uses German) for the month portion of date_exp.

NOW( ) (ODBC 1.0)  
Returns current date and time as a timestamp value.

QUARTER(date_exp) (ODBC 1.0)  
Returns the quarter in date_exp as an integer value in the range of 1–4, where 1 represents January 1 through March 31.

SECOND(time_exp) (ODBC 1.0)  
Returns the second based on the second field in time_exp as an integer value in the range of 0–59.

TIMESTAMPADD(interval, timestamp_exp1, timestamp_exp2)  
(ODBC 2.0)  
Returns the timestamp calculated by adding integer_exp intervals of type interval to timestamp_exp. Valid values of interval are the following keywords:

- SQL_TSI_FRAC_SECOND
- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR

where fractional seconds are expressed in billionths of a second. For example, the following SQL statement returns the name of each employee and his or her one-year anniversary date:

SELECT NAME, {fn TIMESTAMPPADD(SQL_TSI_YEAR, 1, HIRE_DATE)} FROM EMPLOYEES

If timestamp_exp is a time value and interval specifies days, weeks, months, quarters, or years, the date portion of timestamp_exp is set to the current date before calculating the resulting timestamp.

If timestamp_exp is a date value and interval specifies fractional seconds, seconds, minutes, or hours, the time portion of timestamp_exp is set to 0 before calculating the resulting timestamp.

An application determines which intervals a data source supports by calling SQLGetInfo with the SQL_TIMEDATE_ADD_INTERVALS option.

TIMESTAMPDIFF(interval, timestamp_exp1, timestamp_exp2)  
(ODBC 2.0)  
Returns the integer number of intervals of type interval by which timestamp_exp2 is greater than timestamp_exp1. Valid values of interval are the following keywords:

- SQL_TSI_FRAC_SECOND
- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR

where fractional seconds are expressed in billionths of a second. For example, the following SQL statement returns the name of each employee and the number of years he or she has been employed:

SELECT NAME, {fn TIMESTAMPPDIFF(SQL_TSI_YEAR, (fn CURDATE())), HIRE_DATE)} FROM EMPLOYEES

If either timestamp expression is a time value and interval specifies days, weeks, months, quarters, or years, the date portion of that timestamp is set to the current date before calculating the difference between the timestamps.
If either timestamp expression is a date value and interval specifies fractional seconds, seconds, minutes, or hours, the time portion of that timestamp is set to 0 before calculating the difference between the timestamps. An application determines which intervals a data source supports by calling SQLGetInfo with the SQL_TIMEDATE_DIFF_INTERVALS option.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEEK(date_exp) (ODBC 1.0)</td>
<td>Returns the week of the year based on the week field in date_exp as an integer value in the range of 1–53.</td>
</tr>
<tr>
<td>YEAR(date_exp) (ODBC 1.0)</td>
<td>Returns the year based on the year field in date_exp as an integer value. The range is data source–dependent.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**System Functions**

The following table lists system functions that are included in the ODBC scalar function set. By calling SQLGetInfo with an information type of SQL_SYSTEM_FUNCTION, an application can determine which system functions are supported by a driver.

Arguments denoted as exp can be the name of a column, the result of another scalar function, or a literal, where the underlying data type could be represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, SQL_DOUBLE, SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP.

Arguments denoted as value can be a literal constant, where the underlying data type can be represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, SQL_DOUBLE, SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP.

Values returned are represented as ODBC data types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE( ) (ODBC 1.0)</td>
<td>Returns the name of the database corresponding to the connection handle. (The name of the database is also available by calling SQLGetConnectOption with the SQL_CURRENT_QUALIFIER connection option.)</td>
</tr>
<tr>
<td>IFNULL(exp,value) (ODBC 1.0)</td>
<td>If exp is null, value is returned. If exp is not null, exp is returned. The possible data type or types of value must be compatible with the data type of exp.</td>
</tr>
<tr>
<td>USER( ) (ODBC 1.0)</td>
<td>Returns the user name in the DBMS. (The user name is also available by way of SQLGetInfo by specifying the information type: SQL_USER_NAME.) This can be different than the login name.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**Explicit Data Type Conversion Function**

Explicit data type conversion is specified in terms of SQL data type definitions.

The ODBC syntax for the explicit data type conversion function does not restrict conversions. The validity of specific conversions of one data type to another data type will be determined by each driver-specific implementation. The driver will, as it translates the ODBC syntax into the native syntax, reject those conversions that, although legal in the ODBC syntax, are not supported by the data source. The ODBC function SQLGetInfo, with the conversion options (such as SQL_CONVERT_BIGINT, SQL_CONVERT_BINARY, SQL_CONVERT_INTERVAL_YEAR_MONTH, and so on), provides a way to inquire about conversions supported by the data source.

The format of the CONVERT function is:

CONVERT(value_exp, data_type)

The function returns the value specified by value_exp converted to the specified data_type, where data_type is one of the following keywords:

<table>
<thead>
<tr>
<th>data_type</th>
<th>data_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_BIGINT</td>
<td>SQL_INTERVAL_HOUR_TO_MINUTE</td>
</tr>
<tr>
<td>SQL_BINARY</td>
<td>SQL_INTERVAL_HOUR_TO_SECOND</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>SQL_INTERVAL_MINUTE_TO_SECOND</td>
</tr>
<tr>
<td>SQL_CHAR</td>
<td>SQL_LONGVARBINARY</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>SQL_LONGVARCHAR</td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>SQL_NUMERIC</td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td>SQL_REAL</td>
</tr>
<tr>
<td>SQL_GUID</td>
<td>SQL_SMALLINT</td>
</tr>
</tbody>
</table>
The ODBC syntax for the explicit data type conversion function does not support specification of conversion format. If specification of explicit formats is supported by the underlying data source, a driver must specify a default value or implement format specification.

The argument value_exp can be a column name, the result of another scalar function, or a numeric or string literal. For example:

```sql
{ fn CONVERT( { fn CURDATE() }, SQL_CHAR ) }
```

converts the output of the CURDATE scalar function to a character string.

Because ODBC does not mandate a data type for return values from scalar functions (because the functions are often data source-specific), applications should use the CONVERT scalar function whenever possible to force data type conversion.

The following two examples illustrate the use of the CONVERT function. These examples assume the existence of a table called EMPLOYEES, with an EMPNO column of type SQL_SMALLINT and an EMPNAME column of type SQL_CHAR.

If an application specifies the following SQL statement:

```sql
SELECT EMPNO FROM EMPLOYEES WHERE { fn CONVERT(EMPNO,SQL_CHAR) } LIKE '1'
```

• A driver for ORACLE translates the SQL statement to:

```sql
SELECT EMPNO FROM EMPLOYEES WHERE to_char(EMPNO) LIKE '1'
```

• A driver for SQL Server translates the SQL statement to:

```sql
SELECT EMPNO FROM EMPLOYEES WHERE convert(char,EMPNO) LIKE '1'
```

If an application specifies the following SQL statement:

```sql
SELECT { fn ABS(EMPNO) }, { fn CONVERT(EMPNAME,SQL_SMALLINT) }
FROM EMPLOYEES WHERE EMPNO <> 0
```

• A driver for ORACLE translates the SQL statement to:

```sql
SELECT abs(EMPNO), to_number(EMPNAME) FROM EMPLOYEES WHERE EMPNO <> 0
```

• A driver for SQL Server translates the SQL statement to:

```sql
SELECT abs(EMPNO), convert(smallint, EMPNAME) FROM EMPLOYEES
```
A driver for Ingres translates the SQL statement to:

```sql
SELECT abs(EMPNO), Int2(EMPNAME) FROM EMPLOYEES WHERE EMPNO <> 0
```

**SQL-92 CAST Function**

The `CAST` function defined in SQL-92 is equivalent to the `CONVERT` function defined in ODBC. The syntax of the equivalent functions is as follows:

```sql
{ fn CONVERT (value-exp, data-type) } /* ODBC
CAST (value-exp AS data-type) */ SQL92
```

The SQL-92 `CAST` function mandates which data types can be converted to which other data types. (For more information, see the SQL-92 specification.) The `CAST` function is supported at the FIPS Transitional level.

An application can determine support for the `CAST` function as follows:

1. Call `SQLGetInfo` with the `SQL_SQL_CONFORMANCE` information type. If the return value for the information type is `SQL_SC_FIPS127_2_TRANSITIONAL`, `SQL_SC_SQL92_INTERMEDIATE`, or `SQL_SC_SQL92_FULL`, the `CAST` function is supported.
2. If the return value of the `SQL_SQL_CONFORMANCE` information type is `SQL_SC_ENTRY_LEVEL` or 0, call `SQLGetInfo` with the `SQL_SQL92_VALUE_EXPRESSIONS` information type. If the `SQL_SVE_CAST` bit is set, the `CAST` function is supported.

**Appendix F: ODBC Cursor Library**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

The ODBC cursor library (Odbccr32.dll) supports block scrollable cursors for any driver that complies with the Level 1 API conformance level and can be redistributed by developers with their applications or drivers. The cursor library also supports positioned update and delete statements for result sets generated by `SELECT` statements. Although it supports only static and forward-only cursors, the cursor library satisfies the needs of many applications. Furthermore, it can provide good performance, especially for small-sized to medium-sized result sets, and for applications that do not have good cursor support.

The cursor library is a dynamic-link library (DLL) that resides between the Driver Manager and the driver. When an application calls a function, the Driver Manager calls the function in the cursor library, which either executes the function or calls it in the specified driver. For a given connection, an application specifies whether the cursor library is always used, used if the driver does not support scrollable cursors, or never used.

The cursor library appears as a driver to the Driver Manager. If the cursor library resides between the Driver Manager and an ODBC 2.x driver, the cursor library appears as an ODBC 2.x driver. If the cursor library resides between the Driver Manager and an ODBC 3.x driver, the cursor library appears as an ODBC 3.x driver. The behavior exhibited by the cursor library depends upon the version of the driver it is working with, with the exception of binding offsets, which is supported for both ODBC 2.x and ODBC 3.x drivers.

To implement block cursors in `SQLFetch` and `SQLFetchScroll`, the cursor library repeatedly calls `SQLFetch` in the driver. To implement scrolling, it caches the data it has retrieved in memory and in disk files. When an application requests a new rowset, the cursor library retrieves it as necessary from the driver or the cache.

To implement positioned update and delete statements, the cursor library constructs an `UPDATE` or `DELETE` statement with a `WHERE` clause that specifies the cached value of each bound column in the row. When it executes a positioned update statement, the cursor library updates its cache from the values in the rowset buffers.

For more information about the ODBC cursor library, see the following sections of this appendix:

- Using the ODBC Cursor Library
- Executing Positioned Update and Delete Statements
- Cursor Library Code Example
- Implementation Notes
- ODBC Cursor Library Error Codes
Using the ODBC Cursor Library

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

To use the ODBC cursor library, an application:

1. Calls `SQLSetConnectAttr` with an Attribute of SQL_ATTR_ODBC_CURSORS to specify how the cursor library should be used with a particular connection. The cursor library can be always used (SQL_CUR_USE_ODBC), used only if driver does not support scrollable cursors (SQL_CUR_USE_IF_NEEDED), or never used (SQL_CUR_USE_DRIVER).

2. Calls `SQLConnect`, `SQLDriverConnect`, or `SQLBrowseConnect` to connect to the data source.

3. Calls `SQLSetStmtAttr` to specify the cursor type (SQL_ATTR_CURSOR_TYPE), concurrency (SQL_ATTR_CONCURRENCY), and rowset size (SQL_ATTR_ROW_ARRAY_SIZE). The cursor library supports forward-only and static cursors. Forward-only cursors must be read-only, while static cursors can be read-only or can use optimistic concurrency control comparing values.

4. Allocates one or more rowset buffers and calls `SQLBindCol` one or more times to bind these buffers to result set columns.

5. Generates a result set by executing a `SELECT` statement or a procedure, or by calling a catalog function. If the application will execute positioned update statements, it should execute a `SELECT FOR UPDATE` statement to generate the result set.

6. Calls `SQLFetch` or `SQLFetchScroll` one or more times to scroll through the result set.

The application can change data values in the rowset buffers. To refresh the rowset buffers with data from the cursor library’s cache, an application calls `SQLFetchScroll` with the FetchOrientation argument set to SQL_FETCH_RELATIVE and the FetchOffset argument set to 0.

To retrieve data from an unbound column, the application calls `SQLSetPos` to position the cursor on the desired row. It then calls `SQLGetData` to retrieve the data.

To determine the number of rows that have been retrieved from the data source, the application calls `SQLRowCount`.

Executing Positioned Update and Delete Statements

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

After an application has fetched a block of data with `SQLFetchScroll`, it can update or delete the data in the block. To execute a positioned update or delete, the application:

1. Calls `SQLSetPos` to position the cursor on the row to be updated or deleted.

2. Constructs a positioned update or delete statement with the following syntax:

```sql
UPDATE table-name
    SET column-identifier = {expression | NULL}
    [, column-identifier = {expression | NULL}]
WHERE CURRENT OF cursor-name
```

The easiest way to construct the `SET` clause in a positioned update statement is to use parameter markers for each column to be updated and use `SQLBindParameter` to bind these to the rowset buffers for the row to be updated. In this case, the C data type of the parameter will be the same as the C data type of the rowset buffer.

3. Updates the rowset buffers for the current row if it will execute a positioned update statement. After successfully executing a positioned update statement, the cursor library copies the values from each column in the current row to its cache.

**Caution**

If the application does not correctly update the rowset buffers before executing a positioned update statement, the data in the cache will be incorrect after the statement is executed.
4. Executes the positioned update or delete statement using a different statement than the statement associated with the cursor.

<table>
<thead>
<tr>
<th>Statement used</th>
<th>Value in row status array</th>
<th>Values in rowset buffers</th>
<th>Values in cache buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positioned delete</td>
<td>SQL_ROW_DELETED</td>
<td>Old values</td>
<td>Old values</td>
</tr>
</tbody>
</table>

[1] The application must update the values in the rowset buffers before executing the positioned update statement; after executing the positioned update statement, the cursor library copies the values in the rowset buffers to its cache.

## Cursor Library Code Example

```c
#define ROWS 20
#define STATUS_LEN 6
#define OPENDATE_LEN 11
#define DONE -1

SQLHENV    henv;
SQLHDBC    hdbc;
SQLHSTMT   hstmt1, hstmt2;
SQLRETURN  retcode;
SQLCHAR    szStatus[ROWS][STATUS_LEN], szOpenDate[ROWS][OPENDATE_LEN];
SQLSMALLINT szNewStatus[STATUS_LEN], szNewOpenDate[OPENDATE_LEN];
SQLINTEGER sOrderID[ROWS], sNewOrderID[ROWS];
SQLUINTEGER cbStatus[ROWS], cbOrderID[ROWS], cbOpenDate[ROWS];
SQLINTEGER FetchOrientation, crow, FetchOffset, irowUpdt;
SQLSMALLINT RowStatusArray[ROWS];

SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, SQL_OV_ODBC3, 0);
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

// Specify that the ODBC Cursor Library is always used, then connect.
SQLSetEnvAttr(henv, SQL_ATTR_ODBC_CURSORS, SQL_CUR_USE_ODBC, 0);
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt1);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt2);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    /* Allocate a statement handle for the result set and a statement */
    /* handle for positioned update statements. */
    SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt1);
    SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt2);

    /* Specify an updatable static cursor with 20 rows of data. Set */
    /* the cursor name, execute the SELECT statement, and bind the */
Implementation Notes

Important
This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

This section describes how the ODBC cursor library is implemented. It describes how the cursor library maintains its cache, executes SQL statements, and implements ODBC functions.

This section contains the following topics.

- Cursor Library Cache
- Processing SQL Statements
- ODBC Functions and the Cursor Library

Cursor Library Cache

```c
/* rowset buffers to result columns in column-wise fashion. */
SQLSetStmtAttr(hstmt1, SQL_ATTR_CONCURRENCY, SQL_CONCUR_VALUES, 0);
SQLSetStmtAttr(hstmt1, SQL_ATTR_CURSOR_TYPE, SQL_CURSOR_STATIC, 0);
SQLSetStmtAttr(hstmt1, SQL_ATTR_ROW_ARRAY_SIZE, ROWS, 0);
SQLSetStmtAttr(hstmt1, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);
SQLSetStmtAttr(hstmt1, SQL_ATTR_ROWS_FETCHED_PTR, &crow, 0);
SQLSetCursorName(hstmt1, "ORDERCURSOR", SQL_NTS);
SQLExecDirect(hstmt1, "SELECT ORDERID, OPENDATE, STATUS FROM ORDERS ", SQL_NTS);
SQLBindCol(hstmt1, 1, SQL_C_SSHORT, sOrderID, 0, cbName);
SQLBindCol(hstmt1, 2, SQL_C_CHAR, szOpenDate, OPENDATE_LEN, cbOpenDate);
SQLBindCol(hstmt1, 3, SQL_C_CHAR, szStatus, STATUS_LEN, cbStatus);

/* Fetch the first block of data and display it. Prompt the user */
/* for new data values. If the user supplies new values, update */
/* the rowset buffers, bind them to the parameters in the update */
/* statement, and execute a positioned update on another hstmt. */
/* Prompt the user for how to scroll. Fetch and redisplay data as */
/* needed. */

FetchOrientation = SQL_FETCH_FIRST;
FetchOffset = 0;
do {
    SQLFetchScroll(hstmt1, FetchOrientation, FetchOffset);
    DisplayRows(sOrderID, szOpenDate, szStatus, RowStatusArray);
    if (PromptUpdate(&irowUpdt, &sNewOrderID, szNewOpenDate, szNewStatus)==TRUE){
        strcpy_s((char*)sOrderID[irowUpdt], _countof(sOrderID[irowUpdt]), szNewOrderData);
        strcpy_s((char*)szOpenDate[irowUpdt], _countof(szOpenDate[irowUpdt]), szNewOpenData);
        strcpy_s((char*)szStatus[irowUpdt], _countof(szStatus[irowUpdt]), szNewStatus);
        cbOrderID[irowUpdt] = SQL_NTS;
        cbOpenDate[irowUpdt] = SQL_NTS;
        cbStatus[irowUpdt] = SQL_NTS;
        SQLBindParameter(hstmt2, 1, SQL_PARAM_INPUT, SQL_C_SSHORT, SQL_INTEGER, 0, 0,
                         &sOrderID[irowUpdt], 0, &cbOrderID[irowUpdt]);
        SQLBindParameter(hstmt2, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_TYPE_DATE,
                         OPENDATE_LEN, 0, szOpenDate[irowUpdt], OPENDATE_LEN, &cbOpenDate[irowUpdt]);
        SQLBindParameter(hstmt2, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                         STATUS_LEN, 0, szStatus[irowUpdt], STATUS_LEN, &cbStatus[irowUpdt]);
        SQLExecDirect(hstmt2, "UPDATE EMPLOYEE SET ORDERID = ?, OPENDATE = ?, STATUS = ?" 
                "WHERE CURRENT OF EMPCURSOR", SQL_NTS);
    }
}
while (PromptScroll(&FetchOrientation, &FetchOffset) != DONE)
```
Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

For each row of data in the result set, the cursor library caches the data for each bound column, the length of the data in each bound column, and the status of the row. The cursor library uses the values in the cache both to return through SQLFetch and SQLFetchScroll and to construct searched statements for positioned operations. For more information, see Constructing Searched Statements.

This section contains the following topics:

- Column Data
- Length of Column Data
- Row Status
- Location of Cache

© 2016 Microsoft

Column Data

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

The cursor library creates a buffer in the cache for each data buffer bound to the result set with SQLBindCol. It uses the values in these buffers to construct a WHERE clause when it emulates a positioned update or delete statement. It updates these buffers from the rowset buffers when it fetches data from the data source and when it executes positioned update statements.

When the cursor library updates its cache from the rowset buffers, it transfers the data according to the C data type specified in SQLBindCol. For example, if the C data type of a rowset buffer is SQL_C_SLONG, the cursor library transfers four bytes of data; if it is SQL_C_CHAR and BufferLength is 10, the cursor library transfers 10 bytes of data. The cursor library does not perform any type checking or conversions on the data it transfers.

Note

The cursor library does not update its cache for a column if *StrLen_or_IndPtr in the corresponding rowset buffer is SQL_DATA_AT_EXEC or the result of the SQL_LEN_DATA_AT_EXEC macro.

When it updates a column, a data source blank-pads fixed-length character data and zero-pads fixed-length binary data as necessary. For example, a data source stores "Smith" in a CHAR(10) column as "Smith     ". The cursor library does not blank-pad or zero-pad data in the rowset buffers when it copies this data to its cache after executing a positioned update statement. Therefore, if an application requires that the values in the cursor library's cache are blank-padded or zero-padded, it must blank-pad or zero-pad the values in the rowset buffers before executing a positioned update statement.

© 2016 Microsoft

Length of Column Data

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

The cursor library creates a buffer in the cache for each length/indicator buffer bound to the result set with SQLBindCol. It uses the values in these buffers to construct a WHERE clause when it emulates positioned update or delete statements. It updates these buffers from the rowset buffers when it fetches data from the data source and when it executes positioned update statements.

If the C type of a data buffer is SQL_C_CHAR or SQL_C_BINARY, and the length/indicator value is SQL_NTS, the string length of the data is put into the length/indicator buffer.

Note

The cursor library does not update its cache for a column if *StrLen_or_IndPtr in the corresponding rowset buffer is SQL_DATA_AT_EXEC or the result of the SQL_LEN_DATA_AT_EXEC macro.

© 2016 Microsoft

Row Status
The cursor library creates a buffer in the cache for the row status. The cursor library retrieves values for the row status array (specified with the SQL_ATTR_ROW_STATUS_PTR statement attribute) from this buffer. For each row, the cursor library sets this buffer to:

- SQL_ROW_DELETED when it executes a positioned delete statement on the row.
- SQL_ROW_ERROR when it encounters an error retrieving the row from the data source with SQLFetch.
- SQL_ROW_SUCCESS when it successfully fetches the row from the data source with SQLFetch.
- SQL_ROW_UPDATED when it executes a positioned update statement on the row.

### Location of Cache

The cursor library caches data in memory and in Windows® temporary files. This limits the size of the result set that the cursor library can handle only by available disk space. A temporary file is used when the data to be cached would cross the segment boundary if inserted at the end of the cursor library cache. Instead, the data to be cached is added in place of the last-saved block of data in the cache. The last-saved block of data is saved in a temporary file. If the cursor library terminates abnormally, such as when the power fails, it can leave Windows temporary files on the disk. These are named ~CTTnnnn.tmp and are created in the current directory.

**Note**

If the cursor library in Microsoft® WindowsNT®/Windows2000 attempts to cache data in a temporary file on the current directory while the application is running from a read-only share or a compact disk (such as a Microsoft Foundation Class Library sample), SQLSTATE HY000 (General Error-Unable to create a file buffer) will be returned.

### Processing SQL Statements

The ODBC cursor library passes all SQL statements directly to the driver except the following:

- Positioned update and delete statements
- SELECT FOR UPDATE statements
- Batched SQL statements

To execute positioned update and delete statements and to position the cursor on a row to call SQLGetData for that row, the cursor library constructs a searched statement that identifies the row.

This section contains the following topics:

- Processing Positioned Update and Delete Statements
- Processing SELECT FOR UPDATE Statements
- Processing Batches of SQL Statements
- Constructing Searched Statements
**Processing Positioned Update and Delete Statements**

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

The cursor library supports positioned update and delete statements by replacing the `WHERE CURRENT OF` clause in such statements with a `WHERE` clause that enumerates the values stored in its cache for each bound column. The cursor library passes the newly constructed `UPDATE` and `DELETE` statements to the driver for execution. For positioned update statements, the cursor library then updates its cache from the values in the rowset buffers and sets the corresponding value in the row status array to `SQL_ROW_UPDATED`. For positioned delete statements, it sets the corresponding value in the row status array to `SQL_ROW_DELETED`.

**Caution**

The `WHERE` clause constructed by the cursor library to identify the current row can fail to identify any rows, identify a different row, or identify more than one row. For more information, see **Constructing Searched Statements**, later in this appendix.

Positioned update and delete statements are subject to the following restrictions:

- Positioned update and delete statements can be used only in the following cases: when a `SELECT` statement generated the result set; when the `SELECT` statement did not contain a join, a `UNION` clause, or a `GROUP BY` clause; and when any columns that used an alias or expression in the select list were not bound with `SQLBindCol`.

- If an application prepares a positioned update or delete statement, it must do so after it has called `SQLFetch` or `SQLFetchScroll`. Although the cursor library submits the statement to the driver for preparation, it closes the statement and executes it directly when the application calls `SQLExecute`.

- If the driver supports only one active statement, the cursor library fetches the rest of the result set and then refetches the current rowset from its cache before it executes a positioned update or delete statement. If the application then calls a function that returns metadata in a result set (for example, `SQLNumResultCols` or `SQLDescribeCol`), the cursor library returns an error.

- If a positioned update or delete statement is performed on a column of a table that includes a timestamp column that is automatically updated every time an update is performed, all subsequent positioned update or delete statements will fail if the timestamp column is bound. This occurs because the searched update or delete statement that the cursor library creates will not accurately identify the row to update. The value in the searched statement for the timestamp column will not match the automatically updated value of the timestamp column.

---

**Processing SELECT FOR UPDATE Statements**

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

For maximum interoperability, applications should generate result sets that will be updated with a positioned update statement by executing a `SELECT FOR UPDATE` statement. Although the cursor library does not require this, it is required by most data sources that support positioned update statements.

The cursor library ignores the columns in the `FOR UPDATE` clause of a `SELECT FOR UPDATE` statement; it removes this clause before passing the statement to the driver. In the cursor library, the `SQL_ATTR_CONCURRENCY` statement attribute, along with the restrictions mentioned in the previous section, controls whether the columns in a result set can be updated.

---

**Processing Batches of SQL Statements**

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

The cursor library does not support batches of SQL statements, including `SQL` statements for which the `SQL_ATTR_PARAMSET_SIZE` statement attribute is greater than 1. If an application submits a batch of `SQL` statements to the cursor library, the results are undefined.
Constructing Searched Statements

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

To support positioned update and delete statements, the cursor library constructs a searched UPDATE or DELETE statement from the positioned statement. To support calls to SQLGetData in a block of data, the cursor library constructs a searched SELECT statement to create a result set containing the current row of data. In each of these statements, the WHERE clause enumerates the values stored in the cache for each bound column that returns SQL_PRED_SEARCHABLE or SQL_PRED_BASIC for the SQL_DESC_SEARCHABLE field identifier in SQLColAttribute.

Caution

The WHERE clause constructed by the cursor library to identify the current row can fail to identify any rows, identify a different row, or identify more than one row.

If a positioned update or delete statement affects more than one row, the cursor library updates the row status array only for the row on which the cursor is positioned and returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01001 (Cursor operation conflict). If the statement does not identify any rows, the cursor library does not update the row status array and returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01001 (Cursor operation conflict). An application can call SQLRowCount to determine the number of rows that were updated or deleted.

If the SELECT clause used to position the cursor for a call to SQLGetData identifies more than one row, SQLGetData is not guaranteed to return the correct data. If it does not identify any rows, SQLGetData returns SQL_NO_DATA.

If an application conforms to the following guidelines, the WHERE clause constructed by the cursor library should uniquely identify the current row, except when this is impossible, such as when the data source contains duplicate rows.

- **Bind columns that uniquely identify the row.** If the bound columns do not uniquely identify the row, the WHERE clause constructed by the cursor library might identify more than one row. In a positioned update or delete statement, such a clause might cause more than one row to be updated or deleted. In a call to SQLGetData, such a clause might cause the driver to return data for the wrong row. Binding all the columns in a unique key guarantees that each row is uniquely identified.

- **Allocate data buffers large enough that no truncation occurs.** The cursor library's cache is a copy of the values in the rowset buffers bound to the result set with SQLBindCol. If data is truncated when it is placed in these buffers, it will also be truncated in the cache. A WHERE clause constructed from truncated values might not correctly identify the underlying row in the data source.

- **Specify non-null length buffers for binary C data.** The cursor library allocates length buffers in its cache only if the StrLen_or_IndPtr argument in SQLBindCol is non-null. When the TargetType argument is SQL_C_BINARY, the cursor library requires the length of the binary data to construct a WHERE clause from the data. If there is no length buffer for a SQL_C_BINARY column and the application calls SQLGetData or attempts to execute a positioned update or delete statement, the cursor library returns SQL_ERROR and SQLSTATE SL014 (A positioned request was issued and not all column count fields were buffered).

- **Specify non-null length buffers for nullable columns.** The cursor library allocates length buffers in its cache only if the StrLen_or_IndPtr argument in SQLBindCol is non-null. Because SQL_NULL_DATA is stored in the length buffer, the cursor library assumes that any column for which no length buffer is specified is non-nullable. If no length column is specified for a nullable column, the cursor library constructs a WHERE clause that uses the data value for the column. This clause will not correctly identify the row.

© 2016 Microsoft

ODBC Functions and the Cursor Library

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

When the ODBC cursor library is enabled for a connection, the Driver Manager calls functions in the cursor library instead of in the driver. The cursor library either executes the function or calls it in the specified driver.

This section contains the following topics.

- ODBC Functions Executed by the Cursor Library
- ODBC Functions Not Executed by the Cursor Library
- SQLBindCol (Cursor Library)
- SQLBindParameter (Cursor Library)
- SQLBulkOperations (Cursor Library)
- SQLCloseCursor (Cursor Library)
ODBC Functions Executed by the Cursor Library

| SQLBindCol  | SQLGetStmtOption |
| SQLBindParam | SQLNativeSql |
| SQLBindParameter | SQLNumParams |
| SQLCloseCursor | SQLParamOptions |
| SQLEndTran | SQLRowCount |
| SQLExtendedFetch | SQLSetConnectAttr |
| SQLFetchScroll | SQLSetConnectOption |
| SQLFreeHandle | SQLSetDescField |
| SQLFreeStmt | SQLSetDescRec |
| SQLGetData | SQLSetPos |
| SQLGetDescField | SQLSetScrollOptions |
| SQLGetDescRec | SQLSetStmtAttr |
| SQLGetFunctions | SQLSetStmtOption |

© 2016 Microsoft

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

The cursor library executes the following functions. When an application calls a function in this list, the Driver Manager invokes the cursor library, not the driver. Note that the cursor library may call the driver when executing the function.
ODBC Functions Not Executed by the Cursor Library

Important
This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

The cursor library does not execute the following functions. When an application calls one of these functions, the Driver Manager invokes the driver, not the cursor library.

| SQLFetch   | SQLGetEnvAttr   |
| SQLGetConnectAttr | SQLSetDescRec   |
| SQLGetDiagField    | SQLSetEnvAttr   |
| SQLGetDiagRec      |

SQLBindCol (Cursor Library)

Important
This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

This topic discusses the use of the SQLBindCol function in the cursor library. For general information about SQLBindCol, see SQLBindCol Function.

An application allocates one or more buffers for the cursor library to return the current rowset in. It calls SQLBindCol one or more times to bind these buffers to the result set.

An application can call SQLBindCol to rebind result set columns after it has called SQLExtendedFetch, SQLFetch, or SQLFetchScroll, as long as the C data type, column size, and decimal digits of the bound column remain the same. The application need not close the cursor to rebind columns to different addresses.

The cursor library supports setting the SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute to use bind offsets. (SQLBindCol does not have to be called for this rebinding to occur.) If the cursor library is used with an ODBC 3.x driver, the bind offset is not used when SQLFetch is called. The bind offset is used if SQLFetch is called when the cursor library is used with an ODBC 2.x driver because SQLFetch is then mapped to SQLExtendedFetch.

The cursor library supports calling SQLBindCol to bind the bookmark column.

When working with an ODBC 2.x driver, the cursor library returns SQLSTATE HY090 (Invalid string or buffer length) when SQLBindCol is called to set the buffer length for a bookmark column to a value not equal to 4. When working with an ODBC 3.x driver, the cursor library allows the buffer to be any size.

SQLBindParameter (Cursor Library)

Important
This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

This topic discusses the use of the SQLBindParameter function in the cursor library. For general information about SQLBindParameter, see SQLBindParameter Function.

An application can call SQLBindParameter to rebind parameters, as long as the C data type, column size, and decimal digits of the bound column remain the same.

The cursor library supports setting the SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute to use bind offsets. (SQLBindParameter does not have to be called for this rebinding to occur.)
The cursor library supports binding data-at-execution parameters.

© 2016 Microsoft

SQLBulkOperations and the Cursor Library

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

The cursor library does not support calling SQLBulkOperations. For general information about SQLBulkOperations, see SQLBulkOperations Function.

© 2016 Microsoft

SQLCloseCursor_ODBC

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

This topic discusses the use of the SQLCloseCursor function in the cursor library. For general information about SQLCloseCursor, see SQLCloseCursor Function.

The cursor library does not support calling SQLCloseCursor without an open cursor. Attempting this will return SQLSTATE 24000 (Invalid cursor state). Calling SQLFreeStmt with an Option of SQL_CLOSE when no cursor is open is supported by the cursor library.

© 2016 Microsoft

SQLEndTran (Cursor Library)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

This topic discusses the use of the SQLEndTran function in the cursor library. For general information about SQLEndTran, see SQLEndTran Function.

The cursor library does not support transactions and passes calls to SQLEndTran directly to the driver. However, the cursor library does support the cursor commit and rollback behaviors as returned by the data source with the SQL_CURSOR_ROLLBACK_BEHAVIOR and SQL_CURSOR_COMMIT_BEHAVIOR information types:

- For data sources that preserve cursors across transactions, changes that are rolled back in the data source are not rolled back in the cursor library’s cache. To make the cache match the data in the data source, the application must close and reopen the cursor.
- For data sources that close cursors at transaction boundaries, the cursor library closes the cursors and deletes the caches for all statements on the connection.
- For data sources that delete prepared statements at transaction boundaries, the application must reprepare all prepared statements on the connection before reexecuting them.

© 2016 Microsoft

SQLExtendedFetch (Cursor Library)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.
This topic discusses the use of the `SQLExtendedFetch` function in the cursor library. For general information about `SQLExtendedFetch`, see `SQLExtendedFetch Function`.

The cursor library implements `SQLExtendedFetch` by repeatedly calling `SQLFetch` in the driver.

The cursor library supports calling `SQLExtendedFetch` with a `FetchOrientation` of `SQL_FETCH_BOOKMARK`.

When the cursor library is used, calls to `SQLExtendedFetch` cannot be mixed with calls to either `SQLFetchScroll` or `SQLFetch`.

© 2016 Microsoft

**SQLFetch (Cursor Library)**

---

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

This topic discusses the use of the `SQLFetch` function in the cursor library. For general information about `SQLFetch`, see `SQLFetch Function`.

When the cursor library is used, calls to `SQLFetch` cannot be mixed with calls to either `SQLFetchScroll` or `SQLExtendedFetch`.

If `SQLFetch` is called with `SQL_ATTR_ROW_ARRAY_SIZE` set to a value greater than 1, the cursor library will pass the call to the driver. If the driver is an ODBC 2.x driver, the rowset size will be ignored and the call to `SQLFetch` will return a single row of data.

If the cursor library is used with an ODBC 2.x driver, a bind offset (as defined by the `SQL_ATTR_ROW_BIND_OFFSET_PTR` statement attribute) is not used when `SQLFetch` is called.

When the cursor library is loaded, an application cannot call `SQLFetch` to fetch bookmark columns. The cursor library passes the call to `SQLFetch` through to the driver, but the function calls to enable bookmarks and bind the bookmark column are intercepted by the cursor library.

© 2016 Microsoft

**SQLFetchScroll (Cursor Library)**

---

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

This topic discusses the use of the `SQLFetchScroll` function in the cursor library. For general information about `SQLFetchScroll`, see `SQLFetchScroll Function`.

The cursor library implements `SQLFetchScroll` by repeatedly calling `SQLFetch` in the driver. It transfers the data it retrieves from the driver to the rowset buffers provided by the application. It also caches the data in memory and disk files. When an application requests a new rowset, the cursor library retrieves it as necessary from the driver (if it has not been previously fetched) or the cache (if it has been previously fetched). Finally, the cursor library maintains the status of the cached data and returns this information to the application in the row status array.

When the cursor library is used, calls to `SQLFetchScroll` cannot be mixed with calls to either `SQLFetch` or `SQLExtendedFetch`.

When the cursor library is used, calls to `SQLFetchScroll` are supported both for ODBC 2.x and for ODBC 3.x drivers.

**Rowset Buffers**

The cursor library optimizes the transfer of data from the driver to the rowset buffer provided by the application if:

- The application uses row-wise binding.
- There are no unused bytes between fields in the structure the application declares to hold a row of data.
- The fields in which `SQLFetch` or `SQLFetchScroll` returns the length/indicator for a column follows the buffer for that column and precedes the buffer for the next column. These fields are optional.

When the application requests a new rowset, the cursor library retrieves data from its cache and from the driver as necessary. If the new and old rowsets overlap, the cursor library can optimize its performance by reusing the data from the overlapping sections of the rowset buffers. Therefore, unsaved changes to the rowset buffers are lost unless the new and old rowsets overlap and the changes are in the overlapping sections of the rowset buffers. To save the changes, an application submits a positioned update statement.

Note that the cursor library always refreshes the rowset buffers with data from the cache when an application calls `SQLFetchScroll` with the `FetchOrientation` argument set to `SQL_FETCH_RELATIVE` and the `FetchOffset` argument set to 0.

The cursor library supports calling `SQLSetStmtAttr` with an `Attribute` of `SQL_ATTR_ROW_ARRAY_SIZE` to change the rowset size while a cursor is open. The new rowset size will take effect the next time `SQLFetchScroll` is called.
Result Set Membership

The cursor library retrieves data from the driver only as the application requests it. Depending on the data source and the setting of the SQL_CONCURRENCY statement attribute, this has the following consequences:

- The data retrieved by the cursor library might differ from the data that was available at the time the statement was executed. For example, after the cursor was opened, rows inserted at a point beyond the current cursor position can be retrieved by some drivers.
- The data in the result set might be locked by the data source for the cursor library and therefore be unavailable to other users.

After the cursor library has cached a row of data, it cannot detect changes to that row in the underlying data source (except for positioned updates and deletes operating on the same cursor’s cache). This occurs because, for calls to SQLFetchScroll, the cursor library never refetches data from the data source. Instead, it refetches data from its cache.

Scrolling

The cursor library supports the following fetch types in SQLFetchScroll.

<table>
<thead>
<tr>
<th>Cursor type</th>
<th>Fetch types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward-only</td>
<td>SQL_FETCH_NEXT</td>
</tr>
<tr>
<td>Static</td>
<td>SQL_FETCH_NEXT</td>
</tr>
<tr>
<td></td>
<td>SQL_FETCH_PRIOR</td>
</tr>
<tr>
<td></td>
<td>SQL_FETCH_FIRST</td>
</tr>
<tr>
<td></td>
<td>SQL_FETCH_LAST</td>
</tr>
<tr>
<td></td>
<td>SQL_FETCH_RELATIVE</td>
</tr>
<tr>
<td></td>
<td>SQL_FETCH_ABSOLUTE</td>
</tr>
<tr>
<td></td>
<td>SQL_FETCH_BOOKMARK</td>
</tr>
</tbody>
</table>

Errors

When SQLFetchScroll is called and one of the calls to SQLFetch returns SQL_ERROR, the cursor library proceeds as follows. After it completes these steps, the cursor library continues processing.

1. Calls SQLGetDiagRec to obtain error information from the driver and posts this as a diagnostic record in the Driver Manager.
2. Sets the SQL_DIAG_ROW_NUMBER field in the diagnostic record to the appropriate value.
3. Sets the SQL_DIAG_COLUMN_NUMBER field in the diagnostic record to the appropriate value, if applicable; otherwise, it sets it to 0.
4. Sets the value for the row in error in the row status array to SQL_ROW_ERROR.

After the cursor library has called SQLFetch multiple times in its implementation of SQLFetchScroll, any error or warning returned by one of the calls to SQLFetch will be in a diagnostic record and can be retrieved by a call to SQLGetDiagRec. If the data was truncated when it was fetched, the truncated data will now reside in the cursor library’s cache. Subsequent calls to SQLFetchScroll to scroll to a row with truncated data will return the truncated data, and no warning will be raised because the data is fetched from the cursor library’s cache. To keep track of the length of data returned so that it can determine whether the data returned in a buffer has been truncated, an application should bind the length/indicator buffer.

Bookmark Operations

The cursor library supports calling SQLFetchScroll with a FetchOrientation of SQL_FETCH_BOOKMARK. It also supports specifying an offset in the FetchOffset argument that can be used in the bookmark operation. This is the only bookmark operation the cursor library supports. The cursor library does not support calling SQLBulkOperations.

If the application has set the SQL_ATTR_USE_BOOKMARKS statement attribute and has bound to the bookmark column, the cursor library generates a fixed-length bookmark and returns it to the application. The cursor library creates and maintains the bookmarks that it uses; it does not use bookmarks maintained at the data source. When SQLFetchScroll is called to retrieve a block of data that has already been fetched from the data source, it retrieves the data from the cursor library cache. As a result, the bookmark used in a call to SQLFetchScroll with a FetchOrientation of SQL_FETCH_BOOKMARK must be created and maintained by the cursor library.

Interaction with Other Functions
An application must call `SQLFetch` or `SQLFetchScroll` before it prepares or executes any positioned update or delete statements.

---

**SQLFreeStmt (Cursor Library)**

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

This topic discusses the use of the `SQLFreeStmt` function in the cursor library. For general information about `SQLFreeStmt`, see `SQLFreeStmt Function`.

If an application calls `SQLFreeStmt` with the SQL_UNBIND option after it calls `SQLExtendedFetch`, `SQLFetch`, or `SQLFetchScroll`, the cursor library returns an error. Before it can unbind result set columns, an application must call `SQLCloseCursor` or `SQLFreeStmt` with the SQL_CLOSE option.

---

**SQLGetData (Cursor Library)**

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

This topic discusses the use of the `SQLGetData` function in the cursor library. For general information about `SQLGetData`, see `SQLGetData Function`.

The cursor library implements `SQLGetData` by first constructing a `SELECT` statement with a `WHERE` clause that enumerates the values stored in its cache for each bound column in the current row. It then executes the `SELECT` statement to reselect the row and calls `SQLGetData` in the driver to retrieve the data from the data source (as opposed to the cache).

**Caution**

The `WHERE` clause constructed by the cursor library to identify the current row can fail to identify any rows, identify a different row, or identify more than one row. For more information, see `Constructing Searched Statements`.

If the SQL_ATTR_USE_BOOKMARKS statement attribute is set to SQL_UB_VARIABLE, `SQLGetData` can be called on column 0 to return bookmark data.

Calls to `SQLGetData` are subject to the following restrictions:

- `SQLGetData` cannot be called for forward-only cursors.
- `SQLGetData` can be called only when the following conditions are met: a `SELECT` statement generated the result set; the `SELECT` statement did not contain a join, a `UNION` clause, or a `GROUP BY` clause; and any columns that used an alias or expression in the select list were not bound with `SQLBindCol`.
- If the driver supports only one active statement, the cursor library fetches the rest of the result set before executing the `SELECT` statement and calling `SQLGetData`.

---

**SQLGetDescField and SQLGetDescRec (Cursor Library)**

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

This topic discusses the use of the `SQLGetDescField` and `SQLGetDescRec` functions in the cursor library. For general information about these functions, see `SQLGetDescField Function` and `SQLGetDescRec Function`.

The cursor library executes `SQLGetDescRec` to return metadata for bookmark columns. The cursor library executes `SQLGetDescField` to return the same fields returned by `SQLGetDescRec`, which are SQL_DESC_NAME, SQL_DESC_TYPE, SQL_DESC_DATE_TIME_INTERVAL_CODE, SQL_DESC_OCTET_LENGTH, SQL_DESC_PRECISION, SQL_DESC_SCALE, and SQL_DESC_NULLABLE. For consistency, `SQLGetDescField` also returns SQL_DESC_UNNAMED.

The cursor library executes `SQLGetDescField` when it is called to return the value of the following fields that are set for binding bookmark columns: SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, SQL_DESC_OCTET_LENGTH_PTR, and SQL_DESC_LENGTH.
The cursor library executes `SQLGetDescField` when it is called to return the value of the `SQL_DESC_BIND_OFFSET_PTR`, `SQL_DESC_BIND_TYPE`, `SQL_DESC_ROW_ARRAY_SIZE`, or `SQL_DESC_ROW_STATUS_PTR` field. These fields can be returned for any row, not just the bookmark row.

If an application calls `SQLGetDescField` to return the value of any field other than those mentioned previously, the cursor library passes the call to the driver.

© 2016 Microsoft

**SQLGetFunctions (Cursor Library)**

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.</td>
</tr>
</tbody>
</table>

This topic discusses the use of the `SQLGetFunctions` function in the cursor library. For general information about `SQLGetFunctions`, see [SQLGetFunctions Function](#).

When you call `SQLGetFunctions`, the cursor library returns that it supports `SQLExtendedFetch`, `SQLFetchScroll`, `SQLSetPos`, and `SQLSetScrollOptions`, in addition to the functions supported by the driver.

© 2016 Microsoft

**SQLGetInfo (Cursor Library)**

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.</td>
</tr>
</tbody>
</table>

This topic discusses the use of the `SQLGetInfo` function in the cursor library. For general information about `SQLGetInfo`, see [SQLGetInfo Function](#).

The cursor library returns values for the following values of `InfoType` (| represents a bitwise OR); for all other values of `InfoType`, it calls `SQLGetInfo` in the driver.

<table>
<thead>
<tr>
<th><code>InfoType</code></th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_BOOKMARK_PERSISTENCE</td>
<td>SQL_BP_SCROLL</td>
</tr>
<tr>
<td>SQL_DYNAMIC_CURSOR_ATTRIBUTES1</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DYNAMIC_CURSOR_ATTRIBUTES2</td>
<td>0</td>
</tr>
<tr>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1</td>
<td>SQL_CA1_NEXT</td>
</tr>
<tr>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2</td>
<td>SQL_CA2_READ_ONLY_CONCUR</td>
</tr>
<tr>
<td>SQL_GETDATA_EXTENSIONS</td>
<td>SQL_GD_BLOCK</td>
</tr>
</tbody>
</table>

**Note**

When data is retrieved with `SQLFetchScroll`, `SQLGetData` supports the functionality specified with the `SQL_GD_ANY_COLUMN` and `SQL_GD_BOUND` bitmasks.

| SQL_KEYSET_DRIVEN_CURSOR_ATTRIBUTES1 | 0 |
| SQL_KEYSET_DRIVEN_CURSOR_ATTRIBUTES2 | 0 |
| SQL_LOCK_TYPES[1] | SQL_LCK_NO_CHANGE |
| SQL_STATIC_CURSOR_ATTRIBUTES1 | SQL_CA1_NEXT | SQL_CA1_ABSOLUTE | SQL_CA1_RELATIVE | SQL_CA1_BOOKMARK | SQL_CA1_LOCK_NO_CHANGE | SQL_CA1_POS_POSITION | SQL_CA1_POSITIONED_DELETE | SQL_CA1_POSITIONED_UPDATE | SQL_CA1_SELECT_FOR_UPDATE |
The cursor library implements the same cursor behavior when transactions are committed or rolled back as the data source. That is, committing or rolling back a transaction, either by calling SQLEndTran or by using the SQL_ATTR_AUTOCOMMIT connection attribute, can cause the data source to delete the access plans and close the cursors for all statements on a connection. For more information, see the SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR information types in SQLGetInfo.
SQLNativeSql (Cursor Library)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

This topic discusses the use of the SQLNativeSql function in the cursor library. For general information about SQLNativeSql, see SQLNativeSql Function.

If the driver supports this function, the cursor library calls SQLNativeSql in the driver and passes it the SQL statement. For positioned update, positioned delete, and SELECT FOR UPDATE statements, the cursor library modifies the statement before passing it to the driver.

Note

The cursor library incorrectly returns SQLSTATE 34000 (Invalid cursor name) if the cursor name is invalid in a positioned update or delete statement that is passed in the InStatementText argument of SQLNativeSql. SQLNativeSql is not intended to return syntax errors, which are returned only upon statement preparation or execution.

SQLRowCount (Cursor Library)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

This topic discusses the use of the SQLRowCount function in the cursor library. For general information about SQLRowCount, see SQLRowCount Function.

When an application calls SQLRowCount with the statement associated with the cursor, the cursor library returns the number of rows of data it has retrieved from the driver.

When an application calls SQLRowCount with the statement associated with a positioned update or delete statement, the cursor library returns the number of rows affected by the statement.

When an application calls SQLRowCount after a SELECT statement, the cursor library returns –1.

SQLSetConnectAttr (Cursor Library)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver's cursor functionality.

This topic discusses the use of the SQLSetConnectAttr function in the cursor library. For general information about SQLSetConnectAttr, see SQLSetConnectAttr Function.

An application calls SQLSetConnectAttr with the SQL_ATTR_ODBC_CURSORS attribute to specify whether the cursor library is always used, used if the driver does not support scrollable cursors, or never used. The cursor library assumes that a driver supports scrollable cursors if it returns SQL_CA1_RELATIVE for the SQL_STATIC_CURSOR_ATTRIBUTES1 information type in SQLGetInfo.

The application must call SQLSetConnectAttr to specify the cursor library usage after it calls SQLAllocHandle with a HandleType of SQL_HANDLE_DBC to allocate the connection and before it connects to the data source. If an application calls SQLSetConnectAttr with the SQL_ATTR_ODBC_CURSORS attribute while the connection is still active, the cursor library returns an error.

To set a statement attribute supported by the cursor library for all statements associated with a connection, an application must call SQLSetConnectAttr for that statement attribute after it connects to the data source and before it opens the cursor. If an application calls SQLSetConnectAttr with a statement attribute and a cursor is open on a statement associated with the connection, the statement attribute will not be applied to that statement until the cursor is closed and reopened.

SQLSetDescField and SQLSetDescRec (Cursor Library)
This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

This topic discusses the use of the SQLSetDescField and SQLSetDescRec functions in the cursor library. For general information about these functions, see SQLSetDescField Function and SQLSetDescRec Function.

The cursor library executes SQLSetDescField when it is called to return the value of the fields set for bookmark columns:

- SQL_DESC_DATA_PTR
- SQL_DESC_INDICATOR_PTR
- SQL_DESC_OCTET_LENGTH_PTR
- SQL_DESC_LENGTH
- SQL_DESC_OCTET_LENGTH
- SQL_DESC_DATETIME_INTERVAL_CODE
- SQL_DESC_SCALE
- SQL_DESC_PRECISION
- SQL_DESC_TYPE
- SQL_DESC_NAME
- SQL_DESC_UNNAMED
- SQL_DESC_NULLABLE

The cursor library executes SQLSetDescRec for a bookmark column.

When working with an ODBC 2.x driver, the cursor library returns SQLSTATE HY090 (Invalid string or buffer length) when SQLSetDescField or SQLSetDescRec is called to set the SQL_DESC_OCTET_LENGTH field for the bookmark record of an ARD to a value not equal to 4. When working with an ODBC 3.x driver, the cursor library allows the buffer to be any size.

The cursor library executes SQLSetDescField when it is called to return the value of the SQL_DESC_BIND_OFFSET_PTR, SQL_DESC_BIND_TYPE, SQL_DESC_ROW_ARRAY_SIZE, or SQL_DESC_ROW_STATUS_PTR field. These fields can be returned for any row, not just the bookmark row.

The cursor library does not execute SQLSetDescField to change any descriptor field other than the fields mentioned previously. If an application calls SQLSetDescField to set any other field while the cursor library is loaded, the call is passed through to the driver.

The cursor library supports changing the SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR fields of any row of an application row descriptor dynamically (after a call to SQLExtendedFetch, SQLFetch, or SQLFetchScroll). The SQL_DESC_OCTET_LENGTH_PTR field can be changed to a null pointer only to unbind the length buffer for a column.

The cursor library does not support changing the SQL_DESC_BIND_TYPE field in an APD or ARD when a cursor is open. The SQL_DESC_BIND_TYPE field can be changed only after the cursor is closed and before a new cursor is opened. The only descriptor fields that the cursor library supports changing when a cursor is open are SQL_DESC_ARRAY_STATUS_PTR, SQL_DESC_BIND_OFFSET_PTR, SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, SQL_DESC_OCTET_LENGTH_PTR, and SQL_DESC_ROWS_PROCESSED_PTR.

The cursor library does not support modifying the SQL_DESC_COUNT field of the ARD after SQLExtendedFetch or SQLFetchScroll has been called and before the cursor has been closed.

© 2016 Microsoft

SQLSetEnvAttr and the Cursor Library

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Microsoft recommends using the driver’s cursor functionality.

This topic discusses the use of the SQLSetEnvAttr function with the cursor library. For general information about SQLSetEnvAttr, see SQLSetEnvAttr Function.

The cursor library is unaffected by the setting of the SQL_ATTR_ODBC_VERSION environment attribute, regardless of the application version or driver version.

© 2016 Microsoft

SQLSetPos (Cursor Library)
This topic discusses the use of the SQLSetPos function in the cursor library. For general information about SQLSetPos, see SQLSetPos Function.

The cursor library supports the SQL_POSITION operation only for the Operation argument in SQLSetPos. It supports the SQL_LOCK_NO_CHANGE value only for the LockType argument.

If the driver does not support bulk operations, the cursor library returns SQLSTATE HYC00 (Driver not capable) when SQLSetPos is called with RowNumber equal to 0. This driver behavior is not recommended.

The cursor library does not support the SQL_UPDATE and SQL_DELETE operations in a call to SQLSetPos. The cursor library implements a positioned update or delete SQL statement by creating a searched update or delete statement with a WHERE clause that enumerates the values stored in its cache for each bound column. For more information, see Processing Positioned Update and Delete Statements.

If the driver does not support static cursors, an application working with the cursor library should call SQLSetPos only on a rowset fetched by SQLExtendedFetch or SQLFetchScroll, not by SQLFetch. The cursor library implements SQLExtendedFetch and SQLFetchScroll by making repeated calls of SQLFetch (with a rowset size of 1) in the driver. The cursor library passes calls to SQLFetch, on the other hand, through to the driver. If SQLSetPos is called on a multrow rowset fetched by SQLFetch when the driver does not support static cursors, the call will fail because SQLSetPos does not work with forward-only cursors. This will occur even if an application has successfully called SQLSetStmtAttr to set SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_STATIC, which the cursor library supports even if the driver does not support static cursors.

This topic discusses the use of the SQLSetScrollOptions function in the cursor library. For general information about SQLSetScrollOptions, see SQLSetScrollOptions Function.

The cursor library supports SQLSetScrollOptions only for backward compatibility; applications should use the SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, and SQL_ATTR_ROW_ARRAY_SIZE statement attributes instead.

This topic discusses the use of the SQLSetStmtAttr function in the cursor library. For general information about SQLSetStmtAttr, see SQLSetStmtAttr Function.

The cursor library supports the following statement attributes with SQLSetStmtAttr:

- SQL_ATTR_CONCURRENCY
- SQL_ATTR_CURSOR_TYPE
- SQL_ATTR_FETCH_BOOKMARK_PTR
- SQL_ATTR_PARAM_BIND_OFFSET_PTR
- SQL_ATTR_PARAM_BIND_TYPE
- SQL_ATTR_ROW_BIND_OFFSET_PTR
- SQL_ATTR_ROW_BIND_TYPE
- SQL_ATTR_ROW_ARRAY_SIZE
- SQL_ATTR_ROWSET_ARRAY_SIZE
- SQL_ATTR_PARAM_BIND_OFFSET_PTR
- SQL_ATTR_ROWS_PER_FETCH
- SQL_ATTR_USE_BOOKMARKS
- SQL_ATTR_USE_BOOKMARKS

The cursor library supports only the SQL_CURSOR_FORWARD_ONLY and SQL_CURSOR_STATIC values of the SQL_ATTR_CURSOR_TYPE statement attribute.

For forward-only cursors, the cursor library supports the SQL_CONCUR_READ_ONLY value of the SQL_ATTR_CONCURRENCY statement attribute. For static cursors, the cursor library supports the SQL_CONCUR_READ_ONLY and SQL_CONCUR_VALUES values of the SQL_ATTR_CONCURRENCY statement attribute.

The cursor library supports only the SQL_SC_NON_UNIQUE value of the SQL_ATTR_SIMULATE_CURSOR statement attribute.

Although the ODBC specification supports calls to SQLSetStmtAttr with the SQL_ATTR_PARAM_BIND_TYPE or SQL_ATTR_ROW_BIND_TYPE attributes after
SQLFetch or SQLFetchScroll has been called, the cursor library does not. Before it can change the binding type in the cursor library, the application must close the cursor. The cursor library supports changing the SQL_ATTR_ROW_BIND_OFFSET_PTR, SQL_ATTR_PARAM_BIND_OFFSET_PTR, SQL_ATTR_ROWS_FETCHED_PTR, and SQL_ATTR_PARAMS_PROCESSED_PTR statement attributes when a cursor is open.

An application can call SQLSetStmtAttr with an Attribute of SQL_ATTR_ROW_ARRAY_SIZE to change the rowset size while a cursor is open. The new rowset size will take effect the next time SQLFetchScroll or SQLFetch is called.

The cursor library supports setting the SQL_ATTR_PARAM_BIND_OFFSET_PTR or SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute to enable binding offsets. The binding offset will not be used for calls to SQLFetch when the cursor library is used with an ODBC 2.x driver.

The cursor library supports setting the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE.

© 2016 Microsoft

**ODBC Cursor Library Error Codes**

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Description</th>
<th>Can be returned from</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>Cursor is not updatable.</td>
<td>SQLFetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td>01000</td>
<td>Cursor library not used. Load failed.</td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>01000</td>
<td>Cursor library not used. Insufficient driver support.</td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>01000</td>
<td>Cursor library not used. Version mismatch with Driver Manager.</td>
<td>SQLBrowseConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLConnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>01000</td>
<td>Driver returned SQL_SUCCESS_WITH_INFO. The warning message has been lost.</td>
<td>SQLFetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td>S1000</td>
<td>General error: Unable to create file buffer.</td>
<td>SQLFetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetData</td>
</tr>
<tr>
<td>S1000</td>
<td>General error: Unable to read from file buffer.</td>
<td>SQLFetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetData</td>
</tr>
<tr>
<td>S1000</td>
<td>General error: Unable to write to file buffer.</td>
<td>SQLFetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFetchScroll</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLGetData</td>
</tr>
<tr>
<td>S1000</td>
<td>General error: Unable to close or remove file buffer.</td>
<td>SQLFreeHandle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQLFreeStmt</td>
</tr>
</tbody>
</table>

**Important**

This feature will be removed in a future version of Microsoft Data Access Component. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Instead, use driver and server cursors.

The ODBC cursor library returns the following SQLSTATEs in addition to those listed in ODBC API Reference.

**Note**

The cursor library does not order status records; the Driver Manager and ODBC 3.x drivers are responsible for ordering status records.
<table>
<thead>
<tr>
<th>SL001</th>
<th>Positioned request cannot be performed because no searchable columns were bound.</th>
<th>SQLExecDirect</th>
<th>SQLGetData</th>
<th>SQLPrepare</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL002</td>
<td>Positioned request could not be performed because result set was created by a join condition.</td>
<td>SQLExecute</td>
<td>SQLExecDirect</td>
<td>SQLGetData</td>
</tr>
<tr>
<td>SL003</td>
<td>Bound buffer exceeds maximum segment size.</td>
<td>SQLFetch</td>
<td>SQLFetchScroll</td>
<td></td>
</tr>
<tr>
<td>SL004</td>
<td>Result set was not generated by a <code>SELECT</code> statement.</td>
<td>SQLGetData</td>
<td>SQLFetchScroll</td>
<td></td>
</tr>
<tr>
<td>SL005</td>
<td><code>SELECT</code> statement contains a GROUP BY clause.</td>
<td>SQLGetData</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SL006</td>
<td>Parameter arrays are not supported with positioned requests.</td>
<td>SQLPrepare</td>
<td>SQLExecDirect</td>
<td></td>
</tr>
<tr>
<td>SL008</td>
<td>SQLGetData is not allowed on a forward-only (nonbuffered) cursor.</td>
<td>SQLGetData</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SL009</td>
<td>No columns were bound prior to calling <code>SQLFetch</code> or <code>SQLFetchScroll</code>.</td>
<td>SQLFetch</td>
<td>SQLFetchScroll</td>
<td></td>
</tr>
<tr>
<td>SL010</td>
<td><code>SQLBindCol</code> returned SQL_ERROR during an attempt to bind to an internal buffer.</td>
<td>SQLFetch</td>
<td>SQLFetchScroll</td>
<td>SQLGetData</td>
</tr>
<tr>
<td>SL011</td>
<td>Statement option is valid only after calling <code>SQLFetch</code> or <code>SQLFetchScroll</code>.</td>
<td>SQLGetStmtAttr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SL012</td>
<td>Statement bindings may not be changed while a cursor is open.</td>
<td>SQLBindCol</td>
<td>SQLFreeHandle</td>
<td>SQLFreeStmt</td>
</tr>
<tr>
<td>SL014</td>
<td>A positioned request was issued and not all column count fields were buffered.</td>
<td>SQLExecDirect</td>
<td>SQLExecute</td>
<td>SQLPrepare</td>
</tr>
<tr>
<td>SL015</td>
<td><code>SQLFetch</code> and <code>SQLFetchScroll</code> cannot be mixed.</td>
<td>SQLExtendedFetch</td>
<td>SQLFetch</td>
<td>SQLFetchScroll</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

Appendix G: Driver Guidelines for Backward Compatibility

This appendix provides information for driver writers working on ODBC 3.x drivers that need to support ODBC 2.x applications. For more information about backward compatibility, see Backward Compatibility and Standards Compliance.

This section contains the following topics:

- **Block Cursors, Scrollable Cursors, and Backward Compatibility for ODBC 3.x Drivers** — New features are features that exist in ODBC 3.x and not in ODBC 2.x. ODBC 3.x drivers generally do not have to worry about backward compatibility with new features because ODBC 2.x applications never use them. The sole exceptions to this are features related to `SQLFetch`, `SQLFetchScroll`, `SQLSetPos`, and `SQLExtendedFetch`; for more information, see , later in this appendix.

- **Mapping Deprecated Functions** — Duplicated features are features that are implemented differently in ODBC 3.x and ODBC 2.x. ODBC 3.x drivers do not need to worry about backward compatibility with duplicated features because the Driver Manager always maps ODBC 2.x features to ODBC 3.x features when calling an ODBC 3.x driver. Thus, an ODBC 3.x driver sees only ODBC 3.x features. For more information about these mappings, see , later in this appendix.

- **Behavioral Changes and ODBC 3.x Drivers** — Behavior changes are features that are handled differently in ODBC 3.x and ODBC 2.x. ODBC 3.x drivers...
have to worry about behavior changes and act in response to the SQL_ATTR_ODBC_VERSION environment attribute set by the application.

© 2016 Microsoft

Block Cursors, Scrollable Cursors, and Backward Compatibility

The existence of both SQLFetchScroll and SQLExtendedFetch represents the first clear split in ODBC between the Application Programming Interface (API), which is the set of functions the application calls, and the Service Provider Interface (SPI), which is the set of functions the driver implements. This split is necessary so that ODBC 3.x, which uses SQLFetchScroll, be aligned with the standards and also be compatible with ODBC 2.x, which uses SQLExtendedFetch.

The ODBC 3.x API, which is the set of functions the application calls, includes SQLFetchScroll and related statement attributes. The ODBC 3.x SPI, which is the set of functions the driver implements, includes SQLFetchScroll, SQLExtendedFetch, and related statement attributes. Because ODBC does not formally enforce this split between the API and the SPI, it is possible for ODBC 3.x applications to call SQLExtendedFetch and related statement attributes. However, there is no reason for ODBC 3.x application to do this. For more information about APIs and SPIs, see the introduction to ODBC Architecture.

For information about what functions and statement attributes an ODBC 3.x application should use with block and scrollable cursors, see Block Cursors, Scrollable Cursors, and Backward Compatibility for ODBC 3.x Applications.

This section contains the following topics.

- What the Driver Manager Does
- What the Driver Does

© 2016 Microsoft

What the Driver Manager Does

The following table summarizes how the ODBC 3.x Driver Manager maps calls to ODBC 2.x and ODBC 3.x drivers.

<table>
<thead>
<tr>
<th>Function or statement attribute</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_FETCH_BOOKMARK_PTR</td>
<td>Points to the bookmark to use with SQLFetchScroll. The following are implementation details:</td>
</tr>
<tr>
<td></td>
<td>- When an application sets this in an ODBC 2.x driver, the ODBC 3.x Driver Manager caches it. It dereferences the pointer and passes the value to the ODBC 2.x driver in the FetchOffset argument of SQLExtendedFetch when SQLFetchScroll is later called by the application.</td>
</tr>
<tr>
<td></td>
<td>- When an application sets this in an ODBC 3.x driver, the ODBC 3.x Driver Manager passes the call to the driver.</td>
</tr>
<tr>
<td>SQL_ATTR_ROW_STATUS_PTR</td>
<td>Points to the row status array filled by SQLFetch, SQLFetchScroll, SQLBulkOperations, and SQLSetPos. The following are implementation details:</td>
</tr>
<tr>
<td></td>
<td>- When an application sets this in an ODBC 2.x driver, the ODBC 3.x Driver Manager caches its value. It passes this value to the ODBC 2.x driver in the RowStatusArray argument of SQLExtendedFetch when SQLFetchScroll or SQLFetch is called.</td>
</tr>
<tr>
<td></td>
<td>- When an application sets this in an ODBC 3.x driver, the ODBC 3.x Driver Manager passes the call to the driver.</td>
</tr>
<tr>
<td></td>
<td>- In state S6, if an application sets SQL_ATTR_ROW_STATUS_PTR and then calls SQLBulkOperations (with an Operation of SQL_ADD) or SQLSetPos without first calling SQLFetch or SQLFetchScroll, SQLSTATE HY011 (Attribute cannot be set now) is returned.</td>
</tr>
<tr>
<td>SQL_ATTR_ROWS_FETCHED_PTR</td>
<td>Points to the buffer in which SQLFetch and SQLFetchScroll return the number of rows fetched. The following are implementation details:</td>
</tr>
<tr>
<td></td>
<td>- When an application sets this in an ODBC 2.x driver, the ODBC 3.x Driver Manager caches its value. It passes this value to the ODBC 2.x driver in the RowCountPtr argument of SQLExtendedFetch when SQLFetch or SQLFetchScroll is called by the application.</td>
</tr>
<tr>
<td></td>
<td>- When an application sets this in an ODBC 3.x driver, the ODBC 3.x Driver Manager passes the call to the driver.</td>
</tr>
<tr>
<td>SQL_ATTR_ROW_ARRAY_SIZE</td>
<td>Sets the rowset size. The following are implementation details:</td>
</tr>
<tr>
<td></td>
<td>- When an application sets this in an ODBC 2.x driver, the ODBC 3.x Driver Manager maps it to the SQL_ROWSET_SIZE statement attribute.</td>
</tr>
</tbody>
</table>
When an application sets this in an ODBC 3.x driver, the ODBC 3.x Driver Manager passes the call to the driver. When an application working with an ODBC 3.x driver calls SQLSetScrollOptions, SQL_ROWSET_SIZE is set to the value in the RowsetSize argument if the underlying driver does not support SQLSetScrollOptions.

<table>
<thead>
<tr>
<th>SQL RoweSize</th>
<th>Sets the rowset size used by SQLExtendedFetch when SQLExtendedFetch is called by an ODBC 2.x application. The following are implementation details:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• When an application sets this, the ODBC 3.x Driver Manager passes the call to the driver, regardless of driver version.</td>
</tr>
<tr>
<td></td>
<td>• When an application working with an ODBC 2.x driver calls SQLSetScrollOptions, SQL_ROWSET_SIZE is set to the value in the RowsetSize argument.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQLBulkOperations</th>
<th>Performs an insert operation, or update, delete, or fetch by bookmark operations. The following are implementation details:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• When an application calls SQLBulkOperations with an Operation of SQL_ADD in an ODBC 2.x driver, the ODBC 3.x Driver Manager maps it to SQLSetPos with an Operation of SQL_ADD.</td>
</tr>
<tr>
<td></td>
<td>• When working with an ODBC 2.x driver that does not support SQLSetPos with an Operation of SQL_ADD, the ODBC 3.x Driver Manager does not map SQLSetPos with an Operation of SQL_ADD to SQLBulkOperations with an Operation of SQL_ADD. This is because SQLBulkOperations cannot be called in state S7, which in ODBC 2.x was the only state in which SQLSetPos could be called.</td>
</tr>
<tr>
<td></td>
<td>• If the application calls SQLBulkOperations with an Operation of SQL_ADD in an ODBC 2.x driver before calling SQLFetchScroll, the ODBC 3.x Driver Manager returns an error.</td>
</tr>
</tbody>
</table>

| SQLExtendedFetch | Returns the specified rowset. Except for the restriction just noted, the ODBC 3.x Driver Manager passes calls to SQLExtendedFetch to the driver, regardless of the driver version. |

<table>
<thead>
<tr>
<th>SQLFetch</th>
<th>Returns the next rowset. The following are implementation details:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• When an application calls SQLFetch in an ODBC 2.x driver, the ODBC 3.x Driver Manager maps it to SQLExtendedFetch. The FetchOrientation argument of SQLExtendedFetch is set to SQL_FETCH_NEXT. The Driver Manager uses the cached value of the SQL_ATTR_ROW_STATUS_PTR statement attribute for the RowStatusArray argument and the cached value of the SQL_ATTR_ROWS_FETCHED_PTR statement attribute for the RowCountPtr argument.</td>
</tr>
<tr>
<td></td>
<td>• An ODBC 3.x application can mix calls to SQLFetch and SQLFetchScroll in an ODBC 2.x driver because the ODBC 3.x Driver Manager maps SQLFetch to SQLExtendedFetch when an application calls it in an ODBC 2.x driver.</td>
</tr>
<tr>
<td></td>
<td>• If an ODBC 2.x driver does not support SQLExtendedFetch, the ODBC 3.x Driver Manager does not map SQLFetch or SQLFetchScroll to SQLExtendedFetch when an application calls it in that driver. If the application attempts to set SQL_ATTR_ROW_ARRAY_SIZE to a value greater than 1, SQLSTATE HYC00 (Optional feature not implemented) is returned.</td>
</tr>
<tr>
<td></td>
<td>• Except for the restrictions just noted, the ODBC 3.x Driver Manager passes calls to SQLFetch to the driver, regardless of the driver version.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQLFetchScroll</th>
<th>Returns the specified rowset. The following are implementation details:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• When an application calls SQLFetchScroll in an ODBC 2.x driver, the ODBC 3.x Driver Manager maps it to SQLExtendedFetch. It uses the cached value of the SQL_ATTR_ROW_STATUS_PTR statement attribute for the RowStatusArray argument and the cached value of the SQL_ATTR_ROWS_FETCHED_PTR statement attribute for the RowCountPtr argument. If the FetchOrientation argument in SQLFetchScroll is SQL_FETCH_BOOKMARK, it uses the cached value of the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute for the FetchOffset argument and returns an error if the FetchOffset argument of SQLFetchScroll is not 0.</td>
</tr>
<tr>
<td></td>
<td>• When an application calls this in an ODBC 3.x driver, the ODBC 3.x Driver Manager passes the call to the driver.</td>
</tr>
</tbody>
</table>

| SQLSetPos | Performs various positioned operations. The ODBC 3.x Driver Manager passes calls to SQLSetPos to the driver, regardless of the driver version. |

| SQLSetScrollOptions | When the Driver Manager maps SQLSetScrollOptions for an application working with an ODBC 3.x driver that does not support SQLSetScrollOptions, the Driver Manager sets the SQL_ROWSET_SIZE statement option, not the SQL_ATTR_ROW_ARRAY_SIZE statement attribute, to the RowsetSize argument in SQLSetScrollOption. As a result, SQLSetScrollOptions cannot be used by an application when fetching multiple rows by a call to SQLFetch or SQLFetchScroll. It can be used only when fetching multiple rows by a call to SQLExtendedFetch. |

---

**What the Driver Does**

The following table summarizes what functions and statement attributes an ODBC 3.x driver should implement for block and scrollable cursors.
<table>
<thead>
<tr>
<th>Function or statement attribute</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ATTR_ROW_STATUS_PTR</td>
<td>Sets the address of the row status array filled by SQLFetch and SQLFetchScroll. This array is also filled by SQLSetPos if SQLSetPos is called in statement state S6. If SQLSetPos is called in state S7, this array is not filled but the array pointed to by the RowStatusArray argument of SQLExtendedFetch is filled. For more information, see Statement Transitions in Appendix B: ODBC State Transition Tables.</td>
</tr>
<tr>
<td>SQL_ATTR_ROWS_FETCHED_PTR</td>
<td>Sets the address of the buffer in which SQLFetch and SQLFetchScroll return the number of rows fetched. If SQLExtendedFetch is called, this buffer is not filled but the RowCountPtr argument points to the number of rows fetched.</td>
</tr>
<tr>
<td>SQL_ATTR_ROW_ARRAY_SIZE</td>
<td>Sets the rowset size used by SQLFetch and SQLFetchScroll.</td>
</tr>
<tr>
<td>SQL_ROWSET_SIZE</td>
<td>Sets the rowset size used by SQLExtendedFetch. ODBC 3.x drivers implement this if they want to work with ODBC 2.x applications that call SQLExtendedFetch or SQLSetPos.</td>
</tr>
<tr>
<td>SQLBulkOperations</td>
<td>If an ODBC 3.x driver should work with ODBC 2.x applications that use SQLSetPos with an Operation of SQL_ADD, the driver must support SQLSetPos with an Operation of SQL_ADD in addition to SQLBulkOperations with an Operation of SQL_ADD.</td>
</tr>
<tr>
<td>SQLExtendedFetch</td>
<td>Returns the specified rowset. ODBC 3.x drivers implement this if they want to work with ODBC 2.x applications that call SQLExtendedFetch or SQLSetPos. The following are implementation details:</td>
</tr>
<tr>
<td></td>
<td>- The driver retrieves the rowset size from the value of the SQL_ROWSET_SIZE statement attribute.</td>
</tr>
<tr>
<td></td>
<td>- The driver retrieves the address of the row status array from the RowStatusArray statement attribute. The RowStatusArray argument in a call to SQLExtendedFetch must not be a null pointer. (Note that in ODBC 3.x, the SQL_ATTR_ROW_STATUS_PTR statement attribute can be a null pointer.)</td>
</tr>
<tr>
<td></td>
<td>- The driver retrieves the address of the rows fetched buffer from the RowCountPtr argument, not the SQL_ATTR_ROWS_Fetched_PTR statement attribute.</td>
</tr>
<tr>
<td></td>
<td>- The driver returns SQLSTATE 01S01 (Error in row) to indicate that an error has occurred while rows were fetched by a call to SQLExtendedFetch. An ODBC 3.x driver should return SQLSTATE 01S01 (Error in row) only when SQLExtendedFetch is called, not when SQLFetch or SQLFetchScroll is called. To preserve backward compatibility, when SQLSTATE 01S01 (Error in row) is returned by SQLExtendedFetch, the Driver Manager does not order status records in the error queue according to the rules stated in the &quot;Sequence of Status Records&quot; section of SQLGetDiagField.</td>
</tr>
<tr>
<td>SQLFetch</td>
<td>Returns the next rowset. The following are implementation details:</td>
</tr>
<tr>
<td></td>
<td>- The driver retrieves the rowset size from the value of the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.</td>
</tr>
<tr>
<td></td>
<td>- The driver retrieves the address of the row status array from the SQL_ATTR_ROW_STATUS_PTR statement attribute.</td>
</tr>
<tr>
<td></td>
<td>- The driver retrieves the address of the rows fetched buffer from the SQL_ATTR_ROWS_Fetched_PTR statement attribute.</td>
</tr>
<tr>
<td></td>
<td>- The application can mix calls between SQLFetchScroll and SQLFetch.</td>
</tr>
<tr>
<td></td>
<td>- SQLFetch returns bookmarks if column 0 is bound.</td>
</tr>
<tr>
<td></td>
<td>- SQLFetch can be called to return more than one row.</td>
</tr>
<tr>
<td></td>
<td>- The driver does not return SQLSTATE 01S01 (Error in row) to indicate that an error has occurred while rows were fetched by a call to SQLFetch.</td>
</tr>
<tr>
<td>SQLFetchScroll</td>
<td>Returns the specified rowset. The following are implementation details:</td>
</tr>
<tr>
<td></td>
<td>- The driver retrieves the rowset size from the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.</td>
</tr>
<tr>
<td></td>
<td>- The driver retrieves the address of the row status array from the SQL_ATTR_ROW_STATUS_PTR statement attribute.</td>
</tr>
<tr>
<td></td>
<td>- The driver retrieves the address of the rows fetched buffer from the SQL_ATTR_ROWS_Fetched_PTR statement attribute.</td>
</tr>
<tr>
<td></td>
<td>- The application can mix calls between SQLFetchScroll and SQLFetch.</td>
</tr>
<tr>
<td></td>
<td>- The driver does not return SQLSTATE 01S01 (Error in row) to indicate that an error has occurred while rows were fetched by a call to SQLFetchScroll.</td>
</tr>
<tr>
<td>SQLSetPos</td>
<td>Performs various positioned operations. The following are implementation details:</td>
</tr>
<tr>
<td></td>
<td>- This can be called in statement states S6 or S7. For more details, see Statement Transitions in Appendix B: ODBC State Transition Tables.</td>
</tr>
<tr>
<td></td>
<td>- If this is called in statement state S5 or S6, the driver retrieves the rowset size from the SQL_ATTR_ROWS_Fetched_PTR statement attribute and the address of the row status array from the SQL_ATTR_ROW_STATUS_PTR statement attribute.</td>
</tr>
<tr>
<td></td>
<td>- If this is called in statement state S7, the driver retrieves the rowset size from the SQL_ROWSET_SIZE statement attribute and the address of the row status array from the RowStatusArray argument of SQLExtendedFetch.</td>
</tr>
<tr>
<td></td>
<td>- The driver returns SQLSTATE 01S01 (Error in row) only to indicate that an error has occurred while rows were fetched by a call to SQLSetPos.</td>
</tr>
</tbody>
</table>
were fetched by a call to SQLSetPos to perform a bulk operation when the function is called in state S7. To preserve backward compatibility, if SQLSTATE 01S01 (Error in row) is returned by SQLSetPos, the Driver Manager does not order status records in the error queue according to the rules stated in the "Sequence of Status Records" section of SQLGetDiagRec.

- If the driver should work with ODBC 2.x applications that call SQLSetPos with an Operation argument of SQL_ADD, the driver must support SQLSetPos with an Operation argument of SQL_ADD.

Mapping Deprecated Functions

This section describes how deprecated functions are mapped by the ODBC 3.x Driver Manager to guarantee backward compatibility of ODBC 3.x drivers that are used with ODBC 2.x applications. The Driver Manager performs this mapping regardless of the version of the application. Because each of the ODBC 2.x functions in the following list is mapped to the corresponding ODBC 3.x function when called in an ODBC 3.x driver, the ODBC 3.x driver does not have to implement the ODBC 2.x functions.

The mapping in the list is triggered when the driver is an ODBC 3.x driver and the driver does not support the function that is being mapped.

The following table lists all duplicated functionality that was introduced in ODBC 3.x.

<table>
<thead>
<tr>
<th>ODBC 2.x function</th>
<th>ODBC 3.x function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocConnect</td>
<td>SQLAllocHandle</td>
</tr>
<tr>
<td>SQLAllocEnv</td>
<td>SQLAllocHandle</td>
</tr>
<tr>
<td>SQLAllocStmt</td>
<td>SQLAllocHandle</td>
</tr>
<tr>
<td>SQLBindParam[1]</td>
<td>SQLBindParameter</td>
</tr>
<tr>
<td>SQLColAttributes</td>
<td>SQLColAttribute</td>
</tr>
<tr>
<td>SQLError</td>
<td>SQLGetDiagRec</td>
</tr>
<tr>
<td>SQLFreeConnect</td>
<td>SQLFreeHandle</td>
</tr>
<tr>
<td>SQLFreeEnv</td>
<td>SQLFreeHandle</td>
</tr>
<tr>
<td>SQLFreeStmt with an Option of SQL_DROP</td>
<td>SQLFreeHandle</td>
</tr>
<tr>
<td>SQLGetConnectOption</td>
<td>SQLGetConnectAttr</td>
</tr>
<tr>
<td>SQLGetStmtOption</td>
<td>SQLGetStmtAttr</td>
</tr>
<tr>
<td>SQLParamOptions</td>
<td>SQLSetStmtAttr</td>
</tr>
<tr>
<td>SQLSetConnectOption</td>
<td>SQLSetConnectAttr</td>
</tr>
<tr>
<td>SQLSetParam[2]</td>
<td>SQLBindParameter</td>
</tr>
<tr>
<td>SQLSetScrollOption</td>
<td>SQLSetStmtAttr</td>
</tr>
<tr>
<td>SQLSetStmtOption</td>
<td>SQLSetStmtAttr</td>
</tr>
<tr>
<td>SQLTransact</td>
<td>SQLEndTran</td>
</tr>
</tbody>
</table>

[1] Even though this function did not exist in ODBC 2.x, it is in the Open Group and ISO standards.

[2] This is an ODBC 1.0 function.

This section contains the following topics.

- SQLAllocConnect Mapping
- SQLAllocEnv Mapping
- SQLAllocStmt Mapping
- SQLBindParam Mapping
- SQLColAttributes Mapping
SQLAllocConnect Mapping

When an application calls SQLAllocConnect through an ODBC 3.x driver, the call to SQLAllocConnect(henv, phdbc) is mapped to SQLAllocHandle as follows:

1. The Driver Manager allocates a connection and returns it to the application.
2. When the application establishes a connection, the Driver Manager calls

   ```sql
   SQLAllocHandle(SQL_HANDLE_DBC, InputHandle, OutputHandlePtr)
   ```

   in the driver with InputHandle set to henv, and OutputHandlePtr set to phdbc.

SQLAllocEnv Mapping

When an application calls SQLAllocEnv through an ODBC 3.x driver, the call to SQLAllocEnv(phenv) is mapped to SQLAllocHandle as follows:

1. The Driver Manager allocates an environment handle and returns it to the application. The Driver Manager calls SQLSetEnvAttr to set the SQL_ATTR_ODBC_VERSION environment attribute to SQL_OV_ODBC2.
2. When the application establishes the first connection to a driver, the Driver Manager calls

   ```sql
   SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, OutputHandlePtr)
   ```

   in the driver with OutputHandlePtr set to phenv.

SQLAllocStmt Mapping

When an application calls SQLAllocStmt through an ODBC 3.x driver, the call to:

```sql
SQLAllocStmt(hdbc, phstmt)
```
is mapped to **SQLAllocHandle** by the Driver Manager in the driver as follows:

```c
SQLAllocHandle(SQL_HANDLE_STMT, InputHandle, OutputHandlePtr)
```

with `InputHandle` set to `hdbc` and `OutputHandlePtr` set to `phstmt`.

---

**SQLBindParam Mapping**

**SQLBindParam** cannot truly be called deprecated because it was never there in ODBC; however, it still represents duplicated functionality — the Driver Manager needs to export it because ISO and Open Group–compliant applications will be using it. Because **SQLBindParameter** contains all the functionality of **SQLBindParam**, **SQLBindParam** will be mapped on top of **SQLBindParameter** (when the underlying driver is an ODBC 3.x driver). An ODBC 3.x driver does not need to implement **SQLBindParam**.

### Remarks

When the following call to **SQLBindParam** is made:

```c
SQLBindParam( StatementHandle, ParameterNumber, ValueType, ParameterType, ColumnSize, DecimalDigits, ParameterValuePtr )
```

the Driver Manager calls **SQLBindParameter** in the driver as follows:

```c
SQLBindParameter( StatementHandle, ParameterNumber, SQL_PARAM_INPUT, ValueType, ParameterType, ColumnSize, DecimalDigits )
```

See [ODBC 64-Bit Information](https://docs.microsoft.com/en-us/sql/odbc/guides/odbc-64-bit-information), if your application will run on a 64-bit operating system.

---

**See Also**

- [Mapping Deprecated Functions](https://docs.microsoft.com/en-us/sql/odbc/guides/odbc-mapping-deprecated-functions)

---

**SQLColAttributes Mapping**

When an application calls **SQLColAttributes** through an ODBC 3.x driver, the call to **SQLColAttributes** is mapped to **SQLColAttribute** as follows:

```c
SQLColAttribute( StatementHandle, SQL_DESC_NAME, SQL_DESC_NULLABLE, SQL_DESC_COUNT )
```

### Note

The prefix used in `FieldIdentifier` values in ODBC 3.x has been changed from that used in ODBC 2.x. The new prefix is "SQL_DESC"; the old prefix was "SQL_COLUMN".

1. If the application is an ODBC 2.x application, `fDescType` is SQL_COLUMN_TYPE, and the returned type is a concise DATETIME type, the Driver Manager maps the return values for date, time, and timestamp codes.

2. If `fDescType` is SQL_COLUMN_NAME, SQL_COLUMN_NULLABLE, or SQL_COLUMN_COUNT, the Driver Manager calls **SQLColAttribute** in the driver with the `FieldIdentifier` argument mapped to SQL_DESC_NAME, SQL_DESC_NULLABLE, or SQL_DESC_COUNT, as appropriate. All other values of `fDescType` are passed through to the driver.

An ODBC 3.x driver must support all the ODBC 3.x `FieldIdentifiers` listed for **SQLColAttribute**.

An ODBC 3.x driver must support SQL_COLUMN_PRECISION and SQL_DESC_PRECISION, SQL_COLUMN_SCALE and SQL_DESC_SCALE, and SQL_COLUMN_LENGTH and SQL_DESC_LENGTH. These values are different because precision, scale, and length are defined differently in ODBC 3.x than they were in ODBC 2.x. For more information, see [Column Size, Decimal Digits, Transfer Octet Length, and Display Size](https://docs.microsoft.com/en-us/sql/odbc/guides/column-size-decimal-digits-transfer-octet-length-and-display-size) in Appendix D: Data Types.

---

**SQLError Mapping**
When an application calls `SQLError` through an ODBC 3.x driver, the call to

```c
SQLError(henv, hdbc, hstmt, szSqlState, pfnativeError, szErrorMsg, cbErrorMsgMax, pcbErrorMsg)
```

is mapped to

```c
SQLGetDiagRec(HandleType, Handle, RecNumber, szSqlstate, pfnativeErrorPtr, szErrorMsg, cbErrorMsgMax, pcbErrorMsg)
```

with the `HandleType` argument set to the value `SQL_HANDLE_ENV, SQL_HANDLE_DBC, or SQL_HANDLE_STMT`, as appropriate, and the `Handle` argument set to the value in `henv, hdbc, or hstmt`, as appropriate. The `RecNumber` argument is determined by the Driver Manager.

© 2016 Microsoft

### SQLFreeConnect Mapping

When an application calls `SQLFreeConnect` through an ODBC 3.x driver, the call to

```c
SQLFreeConnect(hdbc)
```

is mapped to

```c
SQLFreeHandle(SQL_HANDLE_DBC,Handle)
```

with the `Handle` argument set to the value in `hdbc`.

© 2016 Microsoft

### SQLFreeEnv Mapping

When an application calls `SQLFreeEnv` through an ODBC 3.x driver, the call to

```c
SQLFreeEnv(henv)
```

is mapped to

```c
SQLFreeHandle(SQL_HANDLE_ENV,Handle)
```

with the `Handle` argument set to the value in `henv`.

© 2016 Microsoft

### SQLFreeStmt Mapping

When an application calls `SQLFreeStmt` with an `Option` argument of `SQL_DROP` through an ODBC 3.x driver, the call to

```c
SQLFreeStmt(hstmt, SQL_DROP)
```

is mapped to

```c
SQLFreeHandle(SQL_HANDLE_STMT,Handle)
```

with the `Handle` argument set to the value in `hstmt`.
SQLGetConnectOption Mapping

When an application calls SQLGetConnectOption through an ODBC 3.x driver, the call to

```c
SQLGetConnectOption(hdbc, fOption, pvParam)
```

is mapped as follows:

- If `fOption` indicates an ODBC-defined connection option that returns a string, the Driver Manager calls

```c
SQLGetConnectAttr(ConnectionHandle, Attribute, ValuePtr, BufferLength, NULL)
```

- If `fOption` indicates an ODBC-defined connection option that returns a 32-bit integer value, the Driver Manager calls

```c
SQLGetConnectAttr(ConnectionHandle, Attribute, ValuePtr, 0, NULL)
```

- If `fOption` indicates a driver-defined statement option, the Driver Manager calls

```c
SQLGetConnectAttr(ConnectionHandle, Attribute, ValuePtr, BufferLength, NULL)
```

In the preceding three cases, the `ConnectionHandle` argument is set to the value in `hdbc`, the `Attribute` argument is set to the value in `fOption`, and the `ValuePtr` argument is set to the same value as `pvParam`.

For ODBC-defined string connection options, the Driver Manager sets the `BufferLength` argument in the call to `SQLGetConnectAttr` to the predefined maximum length (`SQL_MAX_OPTION_STRING_LENGTH`); for a nonstring connection option, `BufferLength` is set to 0.

For an ODBC 3.x driver, the Driver Manager no longer checks to see if `Option` is in between SQL_CONN_OPT_MIN and SQL_CONN_OPT_MAX, or is greater than SQL_CONNECT_OPT_DRVR_START. The driver must check the validity of the option values.

© 2016 Microsoft

SQLGetStmtOption Mapping

When an application calls SQLGetStmtOption to an ODBC 3.x driver that does not support it, the call to

```c
SQLGetStmtOption(hstmt, fOption, pvParam)
```

will result as follows:

- If `fOption` indicates an ODBC-defined statement option that returns a string, the Driver Manager calls

```c
SQLGetStmtAttr(StatementHandle, Attribute, ValuePtr, BufferLength, NULL)
```

- If `fOption` indicates an ODBC-defined statement option that returns a 32-bit integer value, the Driver Manager calls

```c
SQLGetStmtAttr(StatementHandle, Attribute, ValuePtr, 0, NULL)
```

- If `fOption` indicates a driver-defined statement option, the Driver Manager calls

```c
SQLGetStmtAttr(StatementHandle, Attribute, ValuePtr, BufferLength, NULL)
```
In the preceding three cases, the StatementHandle argument is set to the value in hstmt, the Attribute argument is set to the value in fOption, and the ValuePtr argument is set to the same value as pvParam.

For ODBC-defined string connection options, the Driver Manager sets the BufferLength argument in the call to SQLGetConnectAttr to the predefined maximum length (SQL_MAX_OPTION_STRING_LENGTH); for a nonstring connection option, BufferLength is set to 0.

The SQL_GET_BOOKMARK statement option has been deprecated in ODBC 3.x. For an ODBC 3.x driver to work with ODBC 2.x applications that use SQL_GET_BOOKMARK, it must support SQL_GET_BOOKMARK. For an ODBC 3.x driver to work with ODBC 2.x applications, it must support setting SQL_USE_BOOKMARKS to SQL_UB_ON and should expose fixed-length bookmarks. If an ODBC 3.x driver supports only variable-length bookmarks, it must return SQLSTATE HYC00 (Optional feature not implemented) if an ODBC 2.x application attempts to set SQL_USE_BOOKMARKS to SQL_UB_ON.

For an ODBC 3.x driver, the Driver Manager no longer checks to see whether Option is in between SQL_STMT_OPT_MIN and SQL_STMT_OPT_MAX, or is greater than SQL_CONNECT_OPT_DRVR_START. The driver must check this.

© 2016 Microsoft

SQLInstallTranslator Mapping

When an ODBC 2.x application calls SQLInstallTranslator through an ODBC 3.x driver, the Driver Manager maps the call to SQLInstallTranslatorEx. An application should not call SQLInstallTranslator in the ODBC 3.x Driver Manager with the lpszInfFile argument set to a value other than NULL. The ODBC.INF file used in ODBC 2.x is no longer supported in ODBC 3.x, even for backward compatibility.

© 2016 Microsoft

SQLParamOptions Mapping

When an application calls SQLParamOptions through an ODBC 3.x driver, the call

```sql
SQLParamOptions(hstmt, crow, piRow);
```

will be mapped to two calls of SQLSetStmtAttr as follows:

```sql
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, crow, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_PROCESSED_PTR, piRow, 0);
```

© 2016 Microsoft

SQLSetConnectOption Mapping

When an ODBC 2.x application calls SQLSetConnectOption through an ODBC 3.x driver, the call to

```sql
SQLSetConnectOption(hdbc, fOption, vParam)
```

will result as follows:

- If fOption indicates an ODBC-defined connection attribute that requires a string, the Driver Manager calls

```sql
SQLSetConnectAttr(ConnectionHandle, Attribute, ValuePtr, SQL_NTS)
```

- If fOption indicates an ODBC-defined connection attribute that returns a 32-bit integer value, the Driver Manager calls
If `fOption` indicates a driver-defined connection attribute, the Driver Manager calls

```
SQLSetConnectAttr(ConnectionHandle, Attribute, ValuePtr, BufferLength)
```

In the preceding three cases, the `ConnectionHandle` argument is set to the value in `hdbc`, the `Attribute` argument is set to the value in `fOption`, and the `ValuePtr` argument is set to the same value as `vParam`.

Because the Driver Manager does not know whether the driver-defined connection attribute needs a string or 32-bit integer value, it has to pass in a valid value for the `BufferLength` argument of `SQLSetConnectAttr`. If the driver has defined special semantics for driver-defined connect attributes and needs to be called using `SQLSetConnectOption`, it must support `SQLSetConnectOption`.

If an ODBC 2.x application calls `SQLSetConnectOption` to set a driver-specific statement option in an ODBC 3.x driver, and the option was defined in an ODBC 2.x version of the driver, a new manifest constant should be defined for the option in the ODBC 3.x driver. If the old manifest constant is used in the call to `SQLSetConnectOption`, the Driver Manager will call `SQLSetConnectAttr` with the `StringLength` argument set to 0.

For an ODBC 3.x driver, the Driver Manager no longer checks to see if `fOption` is in between `SQL_CONN_OPT_MIN` and `SQL_CONN_OPT_MAX`, or is greater than `SQL_CONNECT_OPT_DRV_R_START`.

Setting Statement Options on the Connection Level

In ODBC 2.x, an application could call `SQLSetConnectOption` to set a statement option. When that is done, the driver establishes the statement option as a default for any statements later allocated for that connection. It is driver-defined whether the driver sets the statement option for any existing statements associated with the specified connection.

This ability has been deprecated in ODBC 3.x. ODBC 3.x drivers need only support setting ODBC 2.x statement attributes at the connection level if they want to work with ODBC 2.x applications that use this. ODBC 3.x applications should never set statement attributes at the connection level. ODBC 3.x statement attributes cannot be set at the connection level, with the exception of the `SQL_ATTR_METADATA_ID` and `SQL_ATTR_ASYNC_ENABLE` attributes, which are both connection attributes and statement attributes, and can be set at either the connection level or the statement level.

© 2016 Microsoft

SQLSetParam Mapping

`SQLSetParam` continues to be mapped on top of `SQLBindParameter` as in ODBC 2.x. Even though it is conceptually similar to `SQLBindParam`, the Driver Manager does not map `SQLSetParam` to `SQLBindParam`. This is because certain existing ODBC 2.x drivers use the special value of `BufferLength` (`SQL_SETPARAM_VALUE_MAX`) that the Driver Manager generates when it maps `SQLSetParam` on top of `SQLBindParameter` to determine when it is called by a 1.x ODBC application.

A call to

```
SQLSetParam(hstmt, ipar, fCType, fSqlType, cbColDef, lbScale, rgbValue, pcbValue)
```

will result in the following:

```
SQLBindParameter(StatementHandle, ParameterNumber, SQL_PARAM_INPUT_OUTPUT, ValueType, ParameterType, ColumnSize, DecimalDigits,(ParameterValuePtr, pcbValue)
```

© 2016 Microsoft

SQLSetScrollOptions Mapping

When an application calls `SQLSetScrollOptions` through an ODBC 3.x driver and the driver does not support `SQLSetScrollOptions`, the call to

```
SQLSetScrollOptions(StatementHandle, Concurrency, KeysetSize, RowsetSize)
```

will result as follows:

- A call to
with the `InfoType` argument set to one of the values in the following table, depending on the value of the `KeysetSize` argument in `SQLSetScrollOptions`.

<table>
<thead>
<tr>
<th><code>KeysetSize</code> argument</th>
<th><code>InfoType</code> argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_SCROLL_FORWARD_ONLY</td>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2</td>
</tr>
<tr>
<td>SQL_SCROLL_STATIC</td>
<td>SQL_STATIC_CURSOR_ATTRIBUTES2</td>
</tr>
<tr>
<td>SQL_SCROLL_KEYSET_DRIVEN</td>
<td>SQL_KEYSET_CURSOR_ATTRIBUTES2</td>
</tr>
<tr>
<td>SQL_SCROLL_DYNAMIC</td>
<td>SQL_DYNAMIC_CURSOR_ATTRIBUTES2</td>
</tr>
<tr>
<td>A value greater than the <code>RowsetSize</code> argument</td>
<td>SQL_KEYSET_CURSOR_ATTRIBUTES2</td>
</tr>
</tbody>
</table>

If the value of the `KeysetSize` argument is not listed in the preceding table, the call to `SQLSetScrollOptions` returns SQLSTATE S1107 (Row value out of range) and none of the following steps are performed.

The Driver Manager then verifies whether the appropriate bit is set in the `$InfoValuePtr` value returned by the call to `SQLGetInfo`, according to the value of the `Concurrency` argument in `SQLSetScrollOptions`.

<table>
<thead>
<tr>
<th><code>Concurrency</code> argument</th>
<th><code>InfoType setting</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CONCUR_READ_ONLY</td>
<td>SQL_CA2_READ_ONLY_CONCURRENCY</td>
</tr>
<tr>
<td>SQL_CONCUR_LOCK</td>
<td>SQL_CA2_LOCK_CONCURRENCY</td>
</tr>
<tr>
<td>SQL_CONCUR_ROWVER</td>
<td>SQL_CA2_ROWVER_CONCURRENCY</td>
</tr>
<tr>
<td>SQL_CONCUR_VALUES</td>
<td>SQL_CA2_VALUES_CONCURRENCY</td>
</tr>
</tbody>
</table>

If the `Concurrency` argument is not one of the values in the preceding table, the call to `SQLSetScrollOptions` returns SQLSTATE S1108 (Concurrency option out of range) and none of the following steps are performed. If the appropriate bit (as indicated in the preceding table) is not set in `$InfoValuePtr` to one of the values corresponding to the `Concurrency` argument, the call to `SQLSetScrollOptions` returns SQLSTATE 1C00 (Driver not capable) and none of the following steps are performed.

- A call to

```sql
SQLSetStmtAttr(StatementHandle, SQL_ATTR_CURSOR_TYPE, ValuePtr, 0)
```

with `$ValuePtr` set to one of the values in the following table, according to the value of the `KeysetSize` argument in `SQLSetScrollOptions`.

<table>
<thead>
<tr>
<th><code>KeysetSize</code> argument</th>
<th><code>$ValuePtr</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_SCROLL_FORWARD_ONLY</td>
<td>SQL_CURSOR_FORWARD_ONLY</td>
</tr>
<tr>
<td>SQL_SCROLL_STATIC</td>
<td>SQL_CURSOR_STATIC</td>
</tr>
<tr>
<td>SQL_SCROLL_KEYSET_DRIVEN</td>
<td>SQL_CURSOR_KEYSET_DRIVEN</td>
</tr>
<tr>
<td>SQL_SCROLL_DYNAMIC</td>
<td>SQL_CURSOR_DYNAMIC</td>
</tr>
<tr>
<td>A value greater than the <code>RowsetSize</code> argument</td>
<td>SQL_CURSOR_KEYSET_DRIVEN</td>
</tr>
</tbody>
</table>

- A call to

```sql
SQLSetStmtAttr(StatementHandle, SQL_ATTR_CONCURRENCY, ValuePtr, 0)
```

with `$ValuePtr` set to the `Concurrency` argument in `SQLSetScrollOptions`.

- If the `KeysetSize` argument in the call to `SQLSetScrollOptions` is positive, a call to

```sql
SQLSetStmtAttr(StatementHandle, SQL_ATTR_KEYSET_SIZE, ValuePtr, 0)
```
SQLSetStmtOption Mapping

When an application calls SQLSetStmtOption through an ODBC 3.x driver, the call to

\[
\text{SQLSetStmtOption(StatementHandle, fOption, vParam)}
\]

will result as follows:

- If fOption indicates an ODBC-defined statement attribute that is a string, the Driver Manager calls

\[
\text{SQLSetStmtAttr(StatementHandle, fOption, ValuePtr, SQL_NTS)}
\]

- If fOption indicates an ODBC-defined statement attribute that returns a 32-bit integer value, the Driver Manager calls

\[
\text{SQLSetStmtAttr(StatementHandle, fOption, ValuePtr, 0)}
\]

- If fOption indicates a driver-defined statement attribute, the Driver Manager calls

\[
\text{SQLSetStmtAttr(StatementHandle, fOption, ValuePtr, BufferLength)}
\]

In the preceding three cases, the StatementHandle argument is set to the value in hstmt, the Attribute argument is set to the value in fOption, and the ValuePtr argument is set to the value as vParam.

Because the Driver Manager does not know whether the driver-defined statement attribute needs a string or 32-bit integer value, it has to pass in a valid value for the StringLength argument of SQLSetStmtAttr. If the driver has defined special semantics for driver-defined statement attributes and needs to be called using SQLSetStmtOption, it must support SQLSetStmtAttr.

If an application calls SQLSetStmtOption to set a driver-specific statement option in an ODBC 3.x driver and the option was defined in an ODBC 2.x version of the driver, a new manifest constant should be defined for the option in the ODBC 3.x driver. If the old manifest constant is used in the call to SQLSetStmtOption, the Driver Manager will call SQLSetStmtAttr with the StringLength argument set to 0.

When an application calls SQLSetStmtAttr to set SQL_ATTR_USE_BOOKMARKS to SQL_UB_ON in an ODBC 3.x driver, the SQL_ATTR_USE_BOOKMARKS statement attribute is set to SQL_UB_FIXED. SQL_UB.ON is the same constant as SQL_UB_FIXED. The Driver Manager passes SQL_UB_FIXED through to the driver. SQL_UB_FIXED has been deprecated in ODBC 3.x, but an ODBC 3.x driver must implement it to work with ODBC 2.x applications that use fixed-length bookmarks.

For an ODBC 3.x driver, the Driver Manager no longer checks to see if Option is in between SQL_STMT_OPT_MIN and SQL_STMT_OPT_MAX, or is greater than SQL_CONNECT_OPT_DRVR_START.
SQLTransact is now replaced by SQLEndTran. The major difference between the two functions is that SQLEndTran contains an argument HandleType, which specifies the scope of the work to be done. The HandleType argument can specify the environment or the connection handle. The following call to SQLTransact:

```
SQLTransact(henv, hdbc, fType)
```

is mapped to

```
SQLEndTran(SQL_HANDLE_DBC, ConnectionHandle, CompletionType);
```

if ConnectionHandle is not equal to SQL_NULL_HDBC. The ConnectionHandle argument is set to the value of hdbc.

```
SQL_Transact is mapped to
```

```
SQLEndTran (SQL_HANDLE_ENV, EnvironmentHandle, CompletionType);
```

if ConnectionHandle is equal to SQL_NULL_HDBC. The EnvironmentHandle argument is set to the value of henv.

In both of the preceding cases, the CompletionType argument is set to the same value as fType.

© 2016 Microsoft

Behavioral Changes and ODBC 3.x Drivers

The environment attribute SQL_ATTR_ODBC_VERSION indicates to the driver whether it needs to exhibit ODBC 2.x behavior or ODBC 3.x behavior. How the SQL_ATTR_ODBC_VERSION environment attribute is set depends on the application. ODBC 3.x applications must call SQLSetEnvAttr to set this attribute after they call SQLAllocHandle to allocate an environment handle and before they call SQLAllocHandle to allocate a connection handle. If they fail to do this, the Driver Manager returns SQLSTATE HY010 (Function sequence error) on the latter call to SQLAllocHandle.

Note

For more information on behavioral changes and how an application acts, see Behavioral Changes.

ODBC 2.x applications and ODBC 2.x applications recompiled with the ODBC 3.x header files do not call SQLSetEnvAttr. However, they call SQLAllocEnv instead of SQLAllocHandle to allocate an environment handle. Therefore, when the application calls SQLAllocEnv in the Driver Manager, the Driver Manager calls SQLAllocHandle and SQLSetEnvAttr in the driver. Thus, ODBC 3.x drivers can always count on this attribute being set.

If a standards-compliant application compiled with the ODBC_STD compile flag calls SQLAllocEnv (which may occur because SQLAllocEnv is not deprecated in ISO), the call is mapped to SQLAllocHandleStd at compile time. At run time, the application calls SQLAllocHandleStd. The Driver Manager sets the SQL_ATTR_ODBC_VERSION environment attribute to SQL_OV_ODBC3. A call to SQLAllocHandleStd is equivalent to a call to SQLAllocHandle with a HandleType of SQL_HANDLE_ENV and a call to SQLSetEnvAttr to set SQL_ATTR_ODBC_VERSION to SQL_OV_ODBC3.

In certain driver architectures, there is a need for the driver to appear as either an ODBC 2.x driver or an ODBC 3.x driver, depending on the connection. The driver in this case might not actually be a driver but a layer that resides between the Driver Manager and another driver. For example, it might mimic a driver, like ODBC Spy. In another example, it might act as a gateway, like EDA/SQL. To appear as an ODBC 3.x driver, such a driver must be able to export SQLAllocHandle, and to appear as an ODBC 2.x driver, must be able to export SQLAllocConnect, SQLAllocEnv, and SQLAllocStmt. When an environment, connection, or statement is to be allocated, the Driver Manager checks to see if this driver exports SQLAllocHandle. Since the driver does, the Driver Manager calls SQLAllocHandle in the driver. If the driver is working with an ODBC 2.x driver, the driver must map the call to SQLAllocHandle to SQLAllocConnect, SQLAllocEnv, or SQLAllocStmt, as appropriate. It must also do nothing with the SQLSetEnvAttr call when behaving as an ODBC 2.x driver.

This section contains the following topics:

- Datetime Data Types
- Backward Compatibility of C Data Types
- Fixed-Length Bookmarks
- SQLGetInfo Support
- Returning SQL_NO_DATA
- Calling SQLSetPos to Insert Data
- Loading by Ordinal

© 2016 Microsoft
Datetime Data Types

In ODBC 3.x, the identifiers for date, time, and timestamp SQL data types have changed from SQL_DATE, SQL_TIME, and SQL_TIMESTAMP (with instances of `#define` in the header file of 9, 10, and 11) to SQL_TYPE_DATE, SQL_TYPE_TIME, and SQL_TYPE_TIMESTAMP (with instances of `#define` in the header file of 91, 92, and 93), respectively. The corresponding C type identifiers have changed from SQL_C_DATE, SQL_C_TIME, and SQL_C_TIMESTAMP to SQL_C_TYPE_DATE, SQL_C_TYPE_TIME, and SQL_C_TYPE_TIMESTAMP, respectively, and the instances of `#define` have changed accordingly.

The column size and decimal digits returned for the SQL datetime data types in ODBC 3.x are the same as the precision and scale returned for them in ODBC 2.x. These values are different than the values in the SQL_DESC_PRECISION and SQL_DESC_SCALE descriptor fields. (For more information, see Column Size, Decimal Digits, Transfer Octet Length, and Display Size in Appendix D: Data Types.)

These changes affect `SQLDescribeCol`, `SQLDescribeParam`, and `SQLColAttributes`, `SQLBindCol`, `SQLBindParameter`, and `SQLGetData`; and `SQLColumns`, `SQLGetTypeInfo`, `SQLProcedureColumns`, `SQLStatistics`, and `SQLSpecialColumns`.

An ODBC 3.x driver processes the function calls listed in the previous paragraph according to the setting of the `SQL_ATTR_ODBC_VERSION` environment attribute. For `SQLColumns`, `SQLGetTypeInfo`, `SQLProcedureColumns`, `SQLSpecialColumns`, and `SQLStatistics`, if `SQL_ATTR_ODBC_VERSION` is set to `SQL_OV_ODBC3`, the functions return `SQL_TYPE_DATE`, `SQL_TYPE_TIME`, and `SQL_TYPE_TIMESTAMP` in the `DATA_TYPE` field. The `COLUMN_SIZE` column (in the result set returned by `SQLColumns`, `SQLGetTypeInfo`, `SQLProcedureColumns`, and `SQLSpecialColumns`) contains the binary precision for the approximate numeric type. The `NUM_PREC_RADIX` column (in the result set returned by `SQLColumns`, `SQLGetTypeInfo`, and `SQLProcedureColumns`) contains a value of 2. If `SQL_ATTR_ODBC_VERSION` is set to `SQL_OV_ODBC2`, then the functions return `SQL_DATE`, `SQL_TIME`, and `SQL_TIMESTAMP` in the `DATA_TYPE` field, the `COLUMN_SIZE` column contains the decimal precision for the approximate numeric type, and the `NUM_PREC_RADIX` column contains a value of 10.

When all data types are requested in a call to `SQLGetTypeInfo`, the result set returned by the function will contain both `SQL_TYPE_DATE`, `SQL_TYPE_TIME`, and `SQL_TYPE_TIMESTAMP` as defined in ODBC 3.x, and `SQL_DATE`, `SQL_TIME`, and `SQL_TIMESTAMP` as defined in ODBC 2.x.

Because of how the ODBC 3.x Driver Manager performs mapping of the date, time, and timestamp data types, ODBC 3.x drivers need only recognize `#defines` of 91, 92, and 93 for the date, time, and timestamp C data types entered in the `TargetType` arguments of `SQLBindCol` and `SQLGetData` or the `ValueType` argument of `SQLBindParameter` or the `DataType` argument of `SQLGetTypeInfo`. For more information, see Datetime Data Type Changes.

© 2016 Microsoft

Backward Compatibility of C Data Types

`SQL_C_SHORT`, `SQL_C_LONG`, and `SQL_C_TINYINT` have been replaced in ODBC by signed and unsigned types: `SQL_C_SSHORT` and `SQL_C_USHORT`, `SQL_C_SLONG` and `SQL_C_ULONG`, and `SQL_C_STINYINT` and `SQL_C_UTINYINT`. An ODBC 3.x driver that should work with ODBC 2.x applications should support `SQL_C_SHORT`, `SQL_C_LONG`, and `SQL_C_TINYINT`, because when they are called, the Driver Manager passes them through to the driver.

© 2016 Microsoft

Fixed-Length Bookmarks

If an ODBC 3.x driver should work with an ODBC 2.x application that uses fixed-length bookmarks, the driver must support the following:

- `SQL_UB_ON` as a value for the `SQL_USE_BOOKMARKS` statement option. (`SQL_UB_ON` is deprecated in ODBC 3.x.)

- The `SQL_GET_BOOKMARK` statement option.

© 2016 Microsoft

SQLGetInfo Support
When an ODBC 2.x application calls SQLGetInfo to an ODBC 3.x driver, the InfoType arguments in the following table must be supported.

<table>
<thead>
<tr>
<th>InfoType</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ALTER_TABLE (ODBC 2.0)</td>
<td>An SQLINTEGER bitmask enumerating the clauses in the ALTER TABLE statement supported by the data source.</td>
</tr>
<tr>
<td>Note</td>
<td>This information type is not deprecated; the bitmasks in the column to the right are deprecated.</td>
</tr>
<tr>
<td></td>
<td>The following bitmasks are used to determine which clauses are supported:</td>
</tr>
<tr>
<td>SQL_AT_DROP_COLUMN</td>
<td>The ability to drop columns is supported. Whether this results in cascade or restrict behavior is driver-defined. (ODBC 2.0)</td>
</tr>
<tr>
<td>SQL_AT_ADD_COLUMN</td>
<td>The ability to add multiple columns in a single ALTER TABLE statement is supported. This bit does not combine with other SQL_AT_ADD_COLUMN_XXX bits or SQL_AT_CONSTRAINT_XXX bits. (ODBC 2.0)</td>
</tr>
<tr>
<td>SQL_FETCH_DIRECTION (ODBC 1.0)</td>
<td>An SQLINTEGER bitmask enumerating the supported fetch direction options.</td>
</tr>
<tr>
<td></td>
<td>The following bitmasks are used in conjunction with the flag to determine which options are supported:</td>
</tr>
<tr>
<td>SQL_FD_FETCH_NEXT</td>
<td>(ODBC 1.0)</td>
</tr>
<tr>
<td>SQL_FD_FETCH_FIRST</td>
<td>(ODBC 1.0)</td>
</tr>
<tr>
<td>SQL_FD_FETCH_LAST</td>
<td>(ODBC 1.0)</td>
</tr>
<tr>
<td>SQL_FD_FETCH_PRIOR</td>
<td>(ODBC 1.0)</td>
</tr>
<tr>
<td>SQL_FD_FETCH_RELATIVE</td>
<td>(ODBC 1.0)</td>
</tr>
<tr>
<td>SQL_FD_FETCH_ABSOLUTE</td>
<td>(ODBC 1.0)</td>
</tr>
<tr>
<td>SQL_FD_FETCH_BOOKMARK</td>
<td>(ODBC 2.0)</td>
</tr>
<tr>
<td>SQL_LOCK_TYPES (ODBC 2.0)</td>
<td>An SQLINTEGER bitmask enumerating the supported lock types for the fLock argument in SQLSetPos.</td>
</tr>
<tr>
<td></td>
<td>The following bitmasks are used in conjunction with the flag to determine which lock types are supported:</td>
</tr>
<tr>
<td></td>
<td>SQL_LCK_NO_CHANGE SQL_LCK_EXCLUSIVE SQL_LCK_UNLOCK</td>
</tr>
<tr>
<td>SQL_ODBC_API_CONFORMANCE (ODBC 1.0)</td>
<td>An SQLSMALLINT value indicating the level of ODBC conformance.</td>
</tr>
<tr>
<td></td>
<td>SQL_OAC_NONE = None</td>
</tr>
<tr>
<td></td>
<td>SQL_OAC_LEVEL1 = Level 1 supported</td>
</tr>
<tr>
<td></td>
<td>SQL_OAC_LEVEL2 = Level 2 supported</td>
</tr>
<tr>
<td>SQL_ODBC_SQL_CONFORMANCE (ODBC 1.0)</td>
<td>An SQLSMALLINT value indicating SQL grammar supported by the driver. See Appendix C: SQL Grammar for a definition of SQL conformance levels.</td>
</tr>
<tr>
<td></td>
<td>SQL_OSC_MINIMUM = Minimum grammar supported</td>
</tr>
<tr>
<td></td>
<td>SQL_OSC_CORE = Core grammar supported</td>
</tr>
<tr>
<td></td>
<td>SQL_OSC_EXTENDED = Extended grammar supported</td>
</tr>
<tr>
<td>SQL_POS OPERATIONS (ODBC 2.0)</td>
<td>An SQLINTEGER bitmask enumerating the supported operations in SQLSetPos.</td>
</tr>
<tr>
<td></td>
<td>The following bitmasks are used to in conjunction with the flag to determine which options are supported:</td>
</tr>
<tr>
<td>SQL_POS_POSITION</td>
<td>(ODBC 2.0)</td>
</tr>
<tr>
<td>SQL_POS_REFRESH</td>
<td>(ODBC 2.0)</td>
</tr>
<tr>
<td>SQL_POS_UPDATE</td>
<td>(ODBC 2.0)</td>
</tr>
<tr>
<td>SQL_POS_DELETE</td>
<td>(ODBC 2.0)</td>
</tr>
<tr>
<td>SQL_POSITIONED_STATEMENTS (ODBC 2.0)</td>
<td>An SQLINTEGER bitmask enumerating the supported positioned SQL statements.</td>
</tr>
<tr>
<td></td>
<td>The following bitmasks are used to determine which statements are supported:</td>
</tr>
<tr>
<td>SQL_PS_POSITIONED_DELETE</td>
<td>SQL_PS_POSITIONED_UPDATE SQL_PS_SELECT FOR_UPDATE</td>
</tr>
<tr>
<td>SQL_SCROLL_CONCURRENCY (ODBC 1.0)</td>
<td>An SQLINTEGER bitmask enumerating the concurrency control options supported for the cursor.</td>
</tr>
<tr>
<td></td>
<td>The following bitmasks are used to determine which options are supported:</td>
</tr>
<tr>
<td>SQL_SCCO_READ_ONLY</td>
<td>Cursor is read-only. No updates are allowed.</td>
</tr>
<tr>
<td>SQL_SCCO_LOCK</td>
<td>Cursor uses the lowest level of locking sufficient to ensure that the row can be updated.</td>
</tr>
<tr>
<td>SQL_SCCO_OPT_ROWVER</td>
<td>Cursor uses optimistic concurrency control, comparing row versions, such as SQLBase ROWID or Sybase TIMESTAMP.</td>
</tr>
<tr>
<td>SQL_SCCO_OPT_VALUES</td>
<td>Cursor uses optimistic concurrency control, comparing values.</td>
</tr>
<tr>
<td>SQL_STATIC_SENSITIVITY (ODBC 2.0)</td>
<td>An SQLINTEGER bitmask enumerating whether changes made by an application to a static or keyset-driven cursor through SQLSetPos or positioned update or delete statements can be detected by that application.</td>
</tr>
<tr>
<td></td>
<td>SQL_SS_ADDITIONS = Added rows are visible to the cursor; the cursor can scroll to these rows. Where these rows are added to the cursor is driver-dependent.</td>
</tr>
<tr>
<td></td>
<td>SQL_SS_DELETIONS = Deleted rows are no longer available to the cursor and do not leave a &quot;hole&quot; in the result set; after the cursor scrolls from a deleted row, it cannot return to that row.</td>
</tr>
</tbody>
</table>
SQL_SS_UPDATES = Updates to rows are visible to the cursor; if the cursor scrolls from and returns to an updated row, the data returned by the cursor is the updated data, not the original data. This option applies only to static cursors or updates on keyset-driven cursors that do not update the key. This option does not apply for a dynamic cursor or in the case in which a key is changed in a mixed cursor.

Whether an application can detect changes made to the result set by other users, including other cursors in the same application, depends on the cursor type.

An ODBC 3.x application working with an ODBC 3.x driver should not call SQLGetInfo with the InfoType arguments described in the preceding table but should use the ODBC 3.x InfoType arguments listed in the following paragraph. There is not a one-to-one correspondence between InfoType arguments used in ODBC 2.x and those used in ODBC 3.x. An ODBC 3.x application working with an ODBC 2.x driver, on the other hand, should use the InfoType arguments described previously.

Some of the information types in the previous table are deprecated in favor of the cursor attributes information types. These deprecated information types are SQL_FETCH_DIRECTION, SQL_LOCK_TYPES, SQL_POS_OPERATIONS, SQL_POSITIONED_STATEMENTS, SQL_SCROLL_CONCURRENCY, and SQL_STATIC_SENSITIVITY. The new cursor attributes types are SQL_XXX_CURSOR_ATTRIBUTES1 and SQL_XXX_CURSOR_ATTRIBUTES2, where XXX equals DYNAMIC, FORWARD_ONLY, KEYSET_DRIVEN, or STATIC. Each of the new types indicates the driver capabilities for a single cursor type. For more information about these options, see the SQLGetInfo function description.

© 2016 Microsoft

Returning SQL_NO_DATA

When an ODBC 2.x application working with an ODBC 3.x driver calls SQLExecDirect, SQLExecute, or SQLParamData, and a searched update or delete statement was executed but did not affect any rows at the data source, the ODBC 3.x driver should return SQL_SUCCESS. When an ODBC 3.x application working with an ODBC 3.x driver calls SQLExecDirect, SQLExecute, or SQLParamData with the same result, the ODBC 3.x driver should return SQL_NO_DATA.

If a searched update or delete statement in a batch of statements does not affect any rows at the data source, SQLMoreResults returns SQL_SUCCESS. It cannot return SQL_NO_DATA, because that would mean that there are no more results, not that there is a result from a searched update/delete that affected no rows.

© 2016 Microsoft

Calling SQLSetPos to Insert Data

When an ODBC 2.x application working with an ODBC 3.x driver calls SQLSetPos with an Operation argument of SQL_ADD, the Driver Manager does not map this call to SQLBulkOperations. If an ODBC 3.x driver should work with an application that calls SQLSetPos with SQL_ADD, the driver should support that operation.

One major difference in behavior when SQLSetPos is called with SQL_ADD occurs when it is called in state S6. In ODBC 2.x, the driver returned S1010 when SQLSetPos was called with SQL_ADD in state S6 (after the cursor has been positioned with SQLFetch). In ODBC 3.x, SQLBulkOperations with an Operation of SQL_ADD can be called in state S6. A second major difference in behavior is that SQLBulkOperations with an Operation of SQL_ADD can be called in state S5, while SQLSetPos with an Operation of SQL_ADD cannot. For the statement transitions that can occur for the same call in ODBC 3.x, see Appendix B: ODBC State Transition Tables.

© 2016 Microsoft

Loading by Ordinal

In ODBC 2.x, loading by ordinal could be performed to improve the performance of the connection process. An ODBC 2.x driver exports a dummy function with the ordinal 199; when the Driver Manager detects it, it resolves the addresses of the ODBC functions by ordinal, not by name. This functionality is still supported for ODBC 2.x drivers but is not supported for ODBC 3.x drivers.

© 2016 Microsoft

ODBC Data Source Administrator
The Microsoft® ODBC Data Source Administrator manages database drivers and data sources. This application is located in the Windows Control Panel under Administrative Tools. Beginning in Windows 8, the icon is named ODBC Data Sources, and on 64-bit operating systems there is a 32-bit and 64-bit version.

For information about detailed ODBC Administrator procedures, open the ODBC Data Source Administrator dialog box and click Help.

You can use PowerShell commands to manage drivers and data sources. For more information about these PowerShell commands, see Windows Data Access Components PowerShell commands.

This section contains the following topics.

- About Drivers and Data Sources
- Managing Data Sources
- Setting Tracing Options
- Setting ODBC Connection Pooling Options
- Viewing Drivers

© 2016 Microsoft

About Drivers and Data Sources

Drivers are the components that process ODBC requests and return data to the application. If necessary, drivers modify an application’s request into a form that is understood by the data source. You must use the driver’s setup program to add or delete a driver from your computer.

Data sources are the databases or files accessed by a driver and are identified by a data source name (DSN). Use the ODBC Data Source Administrator to add, configure, and delete data sources from your system. The types of data sources that can be used are described in the following table.

<table>
<thead>
<tr>
<th>Data source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>User DSNs are local to a computer and can be used only by the current user. They are registered in the HKEY_CURRENT_USER registry subtree.</td>
</tr>
<tr>
<td>System</td>
<td>System DSNs are local to a computer rather than dedicated to a user. The system or any user with privileges can use a data source set up with a system DSN. System DSNs are registered in the HKEY_LOCAL_MACHINE registry subtree.</td>
</tr>
<tr>
<td>File</td>
<td>File DSNs are file-based sources that can be shared among all users who have the same drivers installed and therefore have access to the database. These data sources need not be dedicated to a user nor be local to a computer. File data source names are not identified by dedicated registry entries; instead, they are identified by a file name with a .dsn extension.</td>
</tr>
</tbody>
</table>

User and system data sources are collectively known as machine data sources because they are local to a computer.

Each of these data sources has a tab in the ODBC Data Source Administrator dialog box. For more information about data sources, see Data Sources.

Managing Data Sources

After you have installed an ODBC driver from the driver’s setup program, you can define one or more data sources for it. The data source name (DSN) should provide a unique description of the data; for example, Payroll or Accounts Payable. The user and system data sources that are defined for all currently installed drivers are listed in the User DSN or System DSN tabs of the ODBC Data Source Administrator dialog box. The file data sources in a given directory are listed in the File DSN tab; the directory to be shown is entered in the Look in box in the File DSN tab.

Note

To manage a data source that connects to a 32-bit driver under 64-bit platform, use c:\windows\sysWOW64\odbcad32.exe. To manage a data source that connects to a 64-bit driver, use c:\windows\system32\odbcad32.exe. In Administrative Tools on a 64-bit Windows 8 operating system, there are icons for both the 32-bit and 64-bit ODBC Data Source Administrator dialog box.

If you use the 64-bit odbcad32.exe to configure or remove a DSN that connects to a 32-bit driver, for example, Driver do Microsoft Access (*.mdb), you will receive the following error message:

The specified DSN contains an architecture mismatch between the Driver and Application

To resolve this error, use the 32-bit odbcad32.exe to configure or remove the DSN.
A data source associates a particular ODBC driver with the data you want to access through that driver. For example, you might create a data source to use the ODBC dBASE driver to access one or more dBASE files found in a specific directory on your hard disk or a network drive. Using the ODBC Data Source Administrator, you can add, modify, and delete data sources, as described in the following table.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding data sources</td>
<td>It is possible to add multiple data sources, each one associating a driver with some data you want to access by using that driver. Give each data source a name that uniquely identifies that data source. For example, if you create a data source for a set of dBASE files that contain customer information, you might name the data source “Customers.” Applications typically display data source names for users to choose from. Adding a file data source is slightly different from adding user or system data sources. For more information, see the ODBC Data Source Administrator help file.</td>
</tr>
<tr>
<td>Modifying data sources</td>
<td>Depending on your requirements, you might find it necessary to reconfigure data sources. You can reset options by clicking Configure in any driver setup dialog box.</td>
</tr>
<tr>
<td>Deleting data sources</td>
<td>Click Remove after selecting a data source.</td>
</tr>
</tbody>
</table>

For more information about file data sources, see Connecting Using File Data Sources or the SQLDriverConnect Function.

See Also

ODBC Data Source Administrator

© 2016 Microsoft

Setting Tracing Options

The Tracing tab of the ODBC Data Source Administrator dialog box enables you to configure the way ODBC function calls are traced.

How Tracing Works

When you start tracing from the Tracing tab, the Driver Manager will log all ODBC function calls for all subsequently run applications. ODBC function calls from applications that are running before tracing is started are not logged. ODBC function calls are recorded in a log file that you specify.

Tracing stops only after you click Stop Tracing Now. Remember that while tracing is on, the log file continues to increase and that this affects the performance of all your ODBC applications.

For more information about tracing, see Tracing.

Changes in ODBC tracing

Prior to MDAC 2.7 SP2, ODBC tracing was only allowed to occur on a machine-wide basis, in which trace captures exposed details about all ODBC applications running under any identities. This included tracing for ODBC-related activity that might occur for processes created or run on behalf of other local user accounts and built-in security principals such as the Local Service and Network Service.

By default, ODBC tracing now uses per-user mode. If you are a local administrator, however, you can still enable machine-wide tracing by using the ODBC Data Source Administrator.

To configure the ODBC tracing mode:

1. If it is necessary, log on using an account that has membership in the Local Administrators' group.
2. From Administrative Tools, open the ODBC Data Source Administrator.
3. Click the Tracing tab.
4. Configure the tracing mode using the Machine-Wide tracing for all user identities check box:
5. To enable machine-wide tracing, select the check box.
6. To return to per-user tracing, clear the check box.
7. Click Apply.

Note
If you have already started tracing in one mode, you have to stop tracing and switch to the other mode for the mode to be changed successfully.

**Important**

Machine-wide tracing should only be enabled when it is needed; otherwise, it should be left turned off.

## Visual Studio Analyzer Tracing

**Important**

Support for Visual Studio Analyzer was removed beginning in Windows 8 (Visual Studio Analyzer was only included in older versions of Visual Studio.). For an alternative troubleshooting mechanism, use BID tracing.

Visual Studio® Analyzer Tracing provides performance and debugging information about the ODBC layer. All outgoing events will be fired at the top-level interface to present as accurate a picture as possible regarding time spent in ODBC components. Visual Studio Analyzer Tracing requires any event source to register when the source is set up. For more information about this kind of tracing, see the Visual Studio documentation.

## Setting ODBC Connection Pooling Options

Connection pooling enables an application to use a connection from a pool of connections that do not need to be reestablished for each use. You can use the Connection Pooling tab of the ODBC Data Source Administrator dialog box to enable and disable performance monitoring. Double-click a driver name to set the connection time-out period.

At the driver level, connection pooling is enabled by the CPTTimeout registry value. This selective per-driver enabling allows a system administrator to enable connection pooling for just the drivers that can support it. It is accomplished by setting the default value of CPTTimeout during the driver’s setup program. Double-click a driver name to set the connection time-out period.

For more information about connection pooling, see [ODBC Connection Pooling](#).

## Performance Monitoring

Performance monitoring tracks connection performance by recording a variety of statistics. These statistics can be customized by the developer to include items such as the following:

<table>
<thead>
<tr>
<th>Counter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBC Hard Connection Counter per Second</td>
<td>The number of actual connections per second that are made to the server. The first time your environment carries a heavy load, this counter will go up very quickly. After a few seconds, it will drop to zero. This is the normal situation when connection pooling is working. When the connections to the server have been established, they will be used and placed in the pool for reuse.</td>
</tr>
<tr>
<td>ODBC Hard Disconnect Counter per Second</td>
<td>The number of hard disconnects per second issued to the server. These are actual connections to the server that are being released by connection pooling. This value will increase from zero when you stop all clients on the system and the connections start to time out.</td>
</tr>
<tr>
<td>ODBC Soft Connection Counter per Second</td>
<td>The number of connections satisfied by the pool per second—in other words, connections from that pool that were handed to users. This counter indicates whether pooling is working. Depending on the load on your server, it is not uncommon for this to show 40–60 soft connections per second.</td>
</tr>
<tr>
<td>ODBC Soft Disconnection Counter per Second</td>
<td>The number of disconnects per second issued by the applications. When the application releases or disconnects, the connection is placed back in the pool.</td>
</tr>
<tr>
<td>ODBC Current Active Connection Counter</td>
<td>The number of connections in the pool that are currently in use.</td>
</tr>
<tr>
<td>ODBC Current Free Connection Counter</td>
<td>The current number of free connections available in the pool. These are live connections that are available for use.</td>
</tr>
<tr>
<td>Pools Currently</td>
<td>The number of pools currently active. This counter was added in Windows 8, for drivers that manage connections in the connection pool. For more information, see <a href="#">Driver-Aware Connection Pooling</a>.</td>
</tr>
</tbody>
</table>
Active

Pools Created
The number of pools active, including active and removed pools. This counter was added in Windows 8, for drivers that manage connections in the connection pool. For more information, see Driver-Aware Connection Pooling.

You must specify your own monitoring parameters. Samples for performance monitoring have been included with this version of ODBC.

© 2016 Microsoft

Viewing Drivers

To configure data sources, you must have installed at least one driver on your system. The Drivers tab in the ODBC Data Source Administrator dialog box lists all drivers installed on your computer, including the name, version, company, file name, and file creation date of each driver. You must use the driver's setup program to add or delete a driver from your system. For more information about modifying drivers, see Managing Data Sources.

© 2016 Microsoft
Microsoft-Supplied ODBC Drivers

Drivers are libraries that implement the functions in the ODBC API. Each driver is specific to a particular database management system (DBMS). In most cases, drivers expose the capabilities of the underlying DBMS; they are not required to implement capabilities not supported by the DBMS. Drivers perform specific tasks and can be classified into two categories: file-based and DBMS-based. For more information about drivers and driver architecture, see Drivers.

This section contains the following topics.

- Using 16-Bit and 32-Bit Applications with 32-Bit Drivers
- Microsoft ODBC Desktop Database Drivers
- ODBC Driver for Oracle
- Visual FoxPro ODBC Driver

© 2016 Microsoft

Using 16-Bit and 32-Bit Applications with 32-Bit Drivers

Important

16-bit application support will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Develop 32-bit or 64-bit applications instead.

With the ODBC data access component, you can use 16-bit and 32-bit applications with 32-bit drivers. The Microsoft® Windows® 95/98 and Microsoft Windows NT®/Windows 2000 operating systems support the following combinations of applications and drivers:

- 16-bit applications with 32-bit drivers
- 32-bit applications with 32-bit drivers

Using a 32-bit application with a 16-bit driver is not supported.

Note

Beginning with the release of ODBC version 3.0, Windows NT 4.0 has been supported.

ODBC includes the ODBC components necessary to support the above configurations by "thunking" dynamic-link libraries (DLLs) to convert 16-bit addresses to 32-bit addresses and vice versa. The Setup program determines which operating system you are using and installs ODBC components required by that system. You can also choose to install the ODBC components used by all systems.

In most cases, porting an application or driver from 16-bit to 32-bit involves five types of changes:

- Changes to message-handling code
- Changes because integers and handles are 32 bits
- Changes in calls to Windows application programming interfaces (APIs)
- Changes to make the driver thread-safe
- Changes to ODBC components

From an application or driver programming standpoint, the major difference between 16-bit and 32-bit ODBC components is that they have different file names. From a system standpoint, the architecture of each application or driver connection is different and the tools used to manage data sources are different.

This section contains the following topics.

- Using 16-Bit Applications with 32-Bit Drivers
- Using 32-Bit Applications with 32-Bit Drivers

© 2016 Microsoft
Using 16-Bit Applications with 32-Bit Drivers

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work and plan to modify applications that currently use this feature. Use 32-bit or 64-bit driver manager instead.

You can run 16-bit applications with 32-bit drivers on your Windows-based system as long as the 32-bit driver does not explicitly call Win32 API functions that create threads. The Windows on Windows (WOW) subsystem runs the applications in 16-bit mode and resolves 16-bit calls to the operating system. ODBC thunking DLLs resolve 16-bit calls from the application to 32-bit drivers. The 16-bit applications use the Windows API, and 32-bit drivers use the Win32 API.

**Architecture**

The following illustration shows how 16-bit applications communicate with 32-bit drivers. Between the 16-bit Driver Manager and the 32-bit drivers are generic thunking DLLs that convert 16-bit ODBC calls to 32-bit ODBC calls.

![Architecture Diagram]

**Note**

Anytime a 16-bit application interacts with a 32-bit driver, the 32-bit Driver Manager always returns "2.0" as the version of ODBC supported by the driver.

**Administration**

You can manage data sources for 32-bit drivers by using the ODBC Data Source Administrator. To open the ODBC Administrator on computers running Microsoft® Windows® 2000, open the Windows Control Panel, double-click **Administrative Tools**, and then double-click **Data Sources (ODBC)**. On computers running previous versions of Microsoft Windows, the icon is named **32-bit ODBC** or simply **ODBC**.

The following illustration shows how a 16-bit application calls a 32-bit driver setup DLL. Between the 16-bit installer DLL and the 32-bit driver setup DLL is a generic thunking DLL that converts 16-bit installer DLL calls to 32-bit installer DLL calls.

![Administration Diagram]

In Windows on Windows (16-bit to 32-bit thunking), an additional thunking DLL named Ds32gt.dll converts 16-bit argument values passed through a 32-bit setup DLL back to 16-bit.

**Components**
The ODBC component of the MDAC 2.8 SP1 SDK includes the following files for running 16-bit applications with 32-bit drivers. These components are in the \Redist directory.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odbc16gt.dll</td>
<td>16-bit ODBC generic thunking DLL</td>
</tr>
<tr>
<td>Odbc32gt.dll</td>
<td>32-bit ODBC generic thunking DLL</td>
</tr>
<tr>
<td>Odbcexp32.dll</td>
<td>32-bit installer DLL</td>
</tr>
<tr>
<td>Odbcad32.exe</td>
<td>32-bit Administrator program</td>
</tr>
<tr>
<td>Odbcinst.hlp</td>
<td>Installer Help file</td>
</tr>
<tr>
<td>Ds16gt.dll</td>
<td>16-bit driver setup generic thunking DLL</td>
</tr>
<tr>
<td>Ctl3d32.dll</td>
<td>32-bit three-dimensional window style library</td>
</tr>
</tbody>
</table>

In addition, the following files along with the 16-bit ODBC 2.10 Driver Manager, which are not part of ODBC 3.51, are required by and should be installed with the 16-bit application.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odbc.dll</td>
<td>16-bit Driver Manager</td>
</tr>
<tr>
<td>Odbcinst.dll</td>
<td>16-bit Installer DLL</td>
</tr>
<tr>
<td>Odbcadm.exe</td>
<td>16-bit ODBC Administrator program</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

Using 32-Bit Applications with 32-Bit Drivers

You can run 32-bit applications with 32-bit drivers. The 32-bit applications and the 32-bit drivers use the Win32® API.

Architecture

The following illustration shows how 32-bit applications communicate with 32-bit drivers. The application calls the 32-bit Driver Manager, which in turn calls 32-bit drivers.

![Illustration of 32-bit application architecture](image)

Important

Do not use the 32-bit thunking installer DLL on WindowsNT/Windows2000. Although it has the same file name as the 32-bit installer DLL, it is a different DLL.

Administration

You can manage data sources for 32-bit drivers by using the ODBC Data Source Administrator. To open the ODBC Administrator on computers running Windows 2000, open the Windows Control Panel, double-click Administrative Tools, and then double-click Data Sources (ODBC). On computers running previous versions of Microsoft Windows, the icon is named 32-bit ODBC or simply ODBC.
Components

The ODBC component includes the following files for running 32-bit applications with 32-bit drivers. These components are in the \Redist directory.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odbc32.dll</td>
<td>32-bit Driver Manager</td>
</tr>
<tr>
<td>Odbccp32.dll</td>
<td>32-bit Installer DLL</td>
</tr>
<tr>
<td>Odbcad32.exe</td>
<td>32-bit ODBC Administrator program</td>
</tr>
<tr>
<td>Odbcinst.hlp</td>
<td>Installer Help file</td>
</tr>
<tr>
<td>Msvcr40.dll</td>
<td>C run-time library</td>
</tr>
</tbody>
</table>

Microsoft ODBC Desktop Database Drivers

ODBC is an API that uses Structured Query Language (SQL) as the database access language. You can access a wide variety of database management systems (DBMSs) with the same ODBC source code that is directly incorporated into an application's source code. With the Microsoft ODBC Desktop Database Drivers, a user of an ODBC-enabled application can open, query, and update a desktop database through the ODBC interface.

The Microsoft ODBC Desktop Database Drivers are a Microsoft Jet-based set of ODBC drivers. Whereas Microsoft ODBC Desktop Database Drivers 2.0 include both 16-bit and 32-bit drivers, versions 3.0 and later include only 32-bit drivers that work on Windows 95 or later, Windows NT Workstation or Server version 4.0, Windows 2000 Professional, or Windows 2000 Server. These drivers provide access to the following types of data sources:

- Microsoft Access
- Microsoft Excel
- Paradox
- dBASE
- Text

See Visual FoxPro ODBC Driver for detailed documentation about the Microsoft Visual FoxPro® ODBC Driver.

Note

Access to other data sources, such as Lotus 1-2-3, Microsoft Exchange, and HTML, is enabled by installable ISAM (IISAM) drivers. For more information about these drivers, see "Accessing External Data" in the Microsoft Jet Database Engine Programmer's Reference. ODBC Desktop Database Drivers 4.0 do not support Btrieve and EMS data formats.

This section contains the following topics.

- Desktop Database Drivers Architecture
- History of the Desktop Database Drivers
- Product Support
- Implementing Desktop Database Drivers
- Microsoft Access Driver Programming Considerations
- Microsoft Excel Driver Programming Considerations
- Paradox Driver Programming Considerations
- dBASE Driver Programming Considerations
- Text File Driver Programming Considerations
- Additional Supported ODBC SQL Grammar
- Limitations
- ODBC Errors
- Supported ODBC API Functions
Desktop Database Drivers Architecture

These drivers are designed for use on Microsoft Windows 95 or later, or Windows NT 4.0 and Windows 2000. Only 32-bit applications are supported on Windows 95 or later; 16-bit and 32-bit applications are supported on Windows NT 4.0 and Windows 2000.

Note

For information about the version of ODBC to be used with these drivers, refer to the ODBC Programmer’s Reference, and past and current release notes. Except for noted areas, these drivers conform to the ODBC Programmer’s Reference.

The ODBC Desktop Database Drivers include 32-bit drivers for Microsoft Access, dBASE, Microsoft Excel, Paradox, and Text. No 16-bit drivers are included. (A driver for Microsoft FoxPro is available separately.)

The application/driver architecture on Windows 95 or later is:

The use of these drivers by 16-bit applications on Windows 95 is not supported.

The application/driver architecture on Windows NT 4.0 and Windows 2000 is:

The Desktop Database Drivers are two-tier drivers. In a two-tier configuration, the driver does not perform the process of parsing, validating, optimizing, and executing the query. Instead, Microsoft Jet performs these tasks. It processes ODBC API calls and acts as an SQL engine. Microsoft Jet has become an integral, inseparable part of the drivers: It is shipped with the drivers and resides with the drivers, even if no other application on the computer uses it.

The Desktop Database Drivers consist of six different drivers — or, more precisely, one driver file (Odbcjt32.dll) that the ODBC Driver Manager uses in six different ways. The DRIVERID flag in the registry entry for a data source determines which driver in Odbcjt32.dll the Driver Manager uses. An application passes this flag in the connection string included in a call to SQLDriverConnect. By default, the flag is the ID of the Microsoft Access driver.

The driver setup file changes the DRIVERID flag at setup time. All drivers except the Microsoft Access driver have an associated setup DLL. When you click Setup in the Microsoft ODBC Data Source Administrator for a data source, the ODBC installer DLL (Odbcinst.dll) loads the setup DLL. The setup DLL exports the ODBC installer function SQLConfigDataSource. If a window handle is passed to SQLConfigDataSource, this function displays a setup window and changes the DRIVERID flag according to the driver selected from the user interface.

When a file is created programmatically, a NULL window handle is passed to SQLConfigDataSource, and the function creates a data source dynamically,
changing the DRIVERID flag according to the lpszDriver argument in the function call.

Odbcj32.dll implements ODBC functions on top of the Microsoft Jet API. There is no direct mapping between ODBC and Microsoft Jet functions, however. Many factors, such as the cursor models and SQL mapping, prevent a direct correlation of the functions.

The ODBC driver resides between the Microsoft Jet engine and the ODBC Driver Manager. Some ODBC functions called by an application are handled by the Driver Manager and not passed to the driver. For these functions, Microsoft Jet never sees the function call because it does not have a direct connection to the Driver Manager.

© 2016 Microsoft

History of the Desktop Database Drivers

The following table shows the Desktop Database Drivers version history.

<table>
<thead>
<tr>
<th>Version</th>
<th>Release Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>August 1993</td>
<td>Used the SIMBA query processor produced by PageAhead Software. SIMBA received ODBC calls and SQL statements, processed them into Microsoft Jet installable ISAM calls, and then called the Microsoft Jet ISAM dispatch layer to load and call the appropriate installable ISAM driver.</td>
</tr>
</tbody>
</table>
| 2.0     | December 1994| Used with ODBC 2.0, which significantly expanded ODBC functionality. The major change in version 2.0 was that the Microsoft Jet database engine replaced the SIMBA query processor. With the Microsoft Jet database engine, the Desktop Database Drivers integrated much more tightly with the Microsoft Jet installable ISAM drivers and Microsoft Access technology. Significant enhancements were:
  - Native support for scrollable cursors.
  - Native support for outer joins, updatable and heterogeneous joins, and transactions.
  - 32-bit versions of the drivers for Microsoft Windows NT. |
| 3.0     | October 1995 | Provided support for Windows 95 and Windows NT Workstation or NT Server 3.51. Only 32-bit drivers were included in this release; the 16-bit drivers for Windows version 3.1 were removed. |
| 3.5     | October 1996 | These drivers were double-byte character set (DBCS)-enabled, were better suited for use with Internet applications than previous versions, and accommodated the use of File data source names (DSNs). The Microsoft Access driver was released in an RISC version for use on Alpha platforms for Windows 95/98 and Windows NT 3.51 and later operating systems. |
| 4.0     | Late 1998    | Provides support for Microsoft Jet Engine Unicode format along with compatibility for ANSI format of earlier versions. |

**Note**
The version 3.5 drivers were designed to work with ODBC2.x. Although they also work with ODBC 3.0, they do not support all ODBC 3.0 features. For more information about how these drivers work with ODBC 3.0, see Backward Compatibility and Standards Compliance.

© 2016 Microsoft

Product Support

Product support for ODBC is provided by Microsoft Product Support Services. Because many Microsoft products use ODBC as a core component and also redistribute ODBC drivers, Microsoft Product Support Services considers ODBC to be a part of each product that incorporates ODBC as a component. Support for ODBC is provided in accordance with the support agreement of that product. See the individual product documentation for more information.

© 2016 Microsoft

Implementing Desktop Database Drivers

Instructions and technical notes about how to implement and set up the Microsoft ODBC Desktop Database Drivers can be found in the ODBC Data Source Administrator online help. This file can be accessed during driver setup routines from the Microsoft ODBC Data Source Administrator.
Files to Use with the Desktop Database Drivers

The ODBC Desktop Database Drivers are used with the Microsoft Jet Database Engine, the installable ISAM files, and Microsoft ODBC files. Not all are required for the redistribution of the product, and not all are core components.

<table>
<thead>
<tr>
<th>For information about</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files used by each Desktop Database Driver</td>
<td>ODBC Driver Files</td>
</tr>
<tr>
<td>Microsoft Jet files used by each Desktop Database Driver</td>
<td>Microsoft Jet Files</td>
</tr>
<tr>
<td>IISAM driver files that provide data access between Microsoft Jet and the other DBMS</td>
<td>Installable ISAM Driver Files</td>
</tr>
</tbody>
</table>

ODBC Component Files

The ODBC Driver files are required by each component of ODBC Desktop Database Drivers 4.0.

Some files are required by more than one component. These files must be redistributed if you intend to redistribute any of the ODBC Desktop Database Drivers with your commercial applications.

These files are installed in the \Windows\System directory for Windows 95 or later, or in the \Windows\System32 directory for Windows NT 3.51 or later and Windows 2000. If ODBC files were previously installed in a different directory, make sure that you use the newer files in the \Windows\System (or System32) directory.

Hardware and Software Requirements (ODBC)

This topic lists requirements for using the ODBC Desktop Database Drivers.

Hardware Requirements

To use the ODBC Desktop Database Drivers, you must have:

- An IBM-compatible personal computer.
Software Requirements

To access data with an ODBC driver, you must have:

- The ODBC driver.
- The 32-bit ODBC Driver Manager, version 3.51 or later (Odbc32.dll).
- Microsoft Windows 95 or later, or Windows NT 4.0 or Windows 2000.
- A stack size of at least 20 KB for an application using a Microsoft ODBC driver.

When using Microsoft Windows NT 4.0 or Windows 2000, the 32-bit driver is thread-safe, but only through the use of a global semaphore that controls access to the driver. Concurrent use of the driver is very limited under Windows NT. All access to the Jet ISAM layer will be single-threaded for all applications using the Microsoft Jet engine.

When running multiple 16-bit applications on Windows on Windows (WOW) on Microsoft Windows NT 4.0, the applications must be run in separate memory spaces. (The same memory space cannot be used because ODBC does not support multiple environments in the same process.) To run an application in a separate memory space, select the application’s icon in the Program Manager, open the File menu and click Properties, and then click Run In Separate Memory Space.

The use of these drivers by 16-bit applications on Windows 95 is not supported.

Driver-Specific Hardware and Software Requirements

- The Microsoft Access and dBASE drivers may require changes in the Autoexec.bat or Config.sys files.

Descriptors and Desktop Database Drivers

A descriptor is a data structure that holds information about either column data or dynamic parameters. SQLGetDescField can be used to retrieve the supported descriptors listed below. Implementation Parameter Descriptors (IPD) are not automatically populated because SQLDescribeParam is not supported. Descriptor fields that are not available through Jet (such as SQL_DESC_BASE_TABLE_NAME) are also not supported.

For more information about Jet-supported descriptor fields, see the Microsoft Jet Database Engine Programmer’s Guide.

For more information about descriptors, see the topics under “Descriptors” in the ODBC Programmer’s Reference.

<table>
<thead>
<tr>
<th>Descriptor fields</th>
<th>Support level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DESC_ALLOC_TYPE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_ARRAY_SIZE</td>
<td>Supported only for ARD</td>
</tr>
<tr>
<td>SQL_DESC_ARRAY_STATUS_PTR</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_BIND_OFFSET_PTR</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_BIND_TYPE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_COUNT</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_ROWS_PROCESSED_PTR</td>
<td>Supported only for ARD</td>
</tr>
<tr>
<td>SQL_DESC_AUTO_UNIQUE_VALUE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_BASE_COLUMN_NAME</td>
<td>Supported (NEW)</td>
</tr>
<tr>
<td>SQL_DESC_BASE_TABLE_NAME</td>
<td>Supported (NEW)</td>
</tr>
<tr>
<td>SQL_DESC_CASE_SENSITIVE</td>
<td>Always FALSE</td>
</tr>
<tr>
<td>Description</td>
<td>Support</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>SQL_DESC_CATALOG_NAME</td>
<td>Not supported</td>
</tr>
<tr>
<td>SQL_DESC_CONCISE_TYPE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_DATA_PTR</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_DATETIME_INTERVAL_CODE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_DATETIME_INTERVAL_PRECISION</td>
<td>Supported for INTERVAL C types</td>
</tr>
<tr>
<td>SQL_DESC_DISPLAY_SIZE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_FIXEDPRECSCALE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_INDICATOR_PTR</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_LABEL</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_LENGTH</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_LITERALPREFIX</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_LITERAL_SUFFIX</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_LOCALTYPE_NAME</td>
<td>Not supported (returns EMPTY string)</td>
</tr>
<tr>
<td>SQL_DESC_NAME</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_NULLABLE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_NUMPREC_RADIX</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_OCTETLENGTH</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_OCTETLENGTH_PTR</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_PARAMETER_TYPE</td>
<td>Only input parameters</td>
</tr>
<tr>
<td>SQL_DESCPRECISION</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESCSCALE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESC_SCHEMA_NAME</td>
<td>Not supported</td>
</tr>
<tr>
<td>SQL_DESCSEARCHABLE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESCTABLE_NAME</td>
<td>Not supported</td>
</tr>
<tr>
<td>SQL_DESCTYPE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESCTYPE_NAME</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESCUNNAMED</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESCUNSIGNED</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DESCUPDATABLE</td>
<td>Supported</td>
</tr>
</tbody>
</table>

Note: Unsupported in versions preceding Jet 4.0

Diagnostics for Desktop Database Drivers

All errors and warnings not checked or partially checked by the Driver Manager are handled by the driver. The driver also maps native errors, or errors returned by the data source, to SQLSTATEs. Each function listed in the ODBC Programmer's Reference contains a "Diagnostics" section that specifies conditions and messages.

Applications call SQLGetDiagRec to retrieve SQLSTATE, native error code, and diagnostic messages. Calling SQLGetDiagField and specifying the field retrieves individual diagnostic fields. The support level of the diagnostic identifiers is listed in the following table.
<table>
<thead>
<tr>
<th>DiagIdentifiers</th>
<th>Support level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_DIA_DYNAMIC_FUNCTION</td>
<td>Not supported</td>
</tr>
<tr>
<td>SQL_DIAG_CLASS_ORIGIN</td>
<td>Supported. Always “ODBC 3.0” for versions 3.0 and later of this driver.</td>
</tr>
<tr>
<td>SQL_DIAG_COLUMN_NUMBER</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DIAG_CURSOR_ROW_COUNT</td>
<td>Not supported</td>
</tr>
<tr>
<td>SQL_DIAG_DYNAMIC_FUNCTION_CODE</td>
<td>Not supported</td>
</tr>
<tr>
<td>SQL_DIAG_MESSAGE_TEXT</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DIAG_NATIVE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DIAG_NUMBER</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DIAG_RETURNCODE</td>
<td>Supported but implemented by the Driver Manager</td>
</tr>
<tr>
<td>SQL_DIAG_ROW_COUNT</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DIAG_ROW_NUMBER</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DIAG_SERVER_NAME</td>
<td>Not supported</td>
</tr>
<tr>
<td>SQL_DIAG_SQLSTATE</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DIAG_SUBCLASS_ORIGIN</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL_DIAG_SPECIAL_COLUMNS</td>
<td>Supported</td>
</tr>
<tr>
<td>ConfigDSNEx</td>
<td></td>
</tr>
<tr>
<td>ConfigDSN</td>
<td></td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**Functions Accepting String Parameters**

All functions that take string parameters will be converted to Unicode. (The “W” form of the function will be exported.) Count of bytes is converted to count of characters for those applicable ODBC APIs. This applies to the following functions:

- SQLConnect
- SQLDriverConnect
- SQLColAttributes
- SQLDescribeCol
- SQLError (replaced by SQLGetDiagField)
- SQLExecDirect
- SQLGetCursorName
- SQLSetCursorName
- SQLGetStmtAttr
- SQLGetInfo
- SQLGetStmtOption (becomes SQLGetStmtAttr)
- SQLSetStmtOption (becomes SQLSetStmtAttr)
- SQLGetConnectOption
- SQLSetConnectOption
- SQLGetTypeInfo
- SQLStatistics
- SQLTables
- SQLNativeSQL
- SQLSpecialColumns
- ConfigDSNEx
- ConfigDSN
SQL-92 Compliance

The ODBC Desktop Database Drivers and the underlying Microsoft Jet engine are not SQL-92 compliant. They support many features that have been defined in SQL-92. Some features supported in the driver are not supported in SQL-92. For more information, see the Microsoft Jet Database Engine Programmer’s Guide. The following are the major differences between the two:

- The SQL used by the Desktop Database Drivers supports more powerful expressions than those specified by SQL-92.
- Different rules apply to the BETWEEN predicate.
- The SQL used by the Desktop Database Drivers and ANSI SQL supports different keywords.

The following SQL-92 features are not supported by Microsoft Jet SQL:

- Security statements, such as GRANT and LOCK.
- DISTINCT with aggregate function references.

The following features are enhancements in the SQL used by the Desktop Database Drivers that are not specified by SQL-92:

- The TRANSFORM statement providing support for crosstab queries.
- Additional aggregate functions (StDev and VarP).

**Note**

The Desktop Database Drivers support the standard ANSI syntax for % (percent) and _ (underscore), not * (asterisk) and ? (question mark).

Base Address of Drivers

Each of the ODBC Desktop Database Drivers has been linked with a base address of 0x04c0000.

Setup DLL

The ODBC Administrator calls the function ConfigDSN when users configure data sources.

Translation DLLs and Desktop Database Drivers
Unicode is a method of software character encoding that treats all characters as having a fixed width of two bytes. This method is used as an alternative to Windows ANSI character encoding, which, because it represents characters in one byte, is limited to 256 characters. Because Unicode can represent over 65,000 characters, it accommodates many languages whose characters are not represented in ANSI encoding.

The ODBC 3.5 (or later) Driver Manager is Unicode-enabled. This affects two major areas: function calls and string data types. The Driver Manager maps function string arguments and string data as required by the application and driver, both of which can be either Unicode-enabled or ANSI-enabled.

The ODBC 3.5 (or later) Driver Manager supports the use of a Unicode driver with both a Unicode application and an ANSI application. It also supports the use of an ANSI driver with an ANSI application. The Driver Manager provides limited Unicode-to-ANSI mapping for a Unicode application working with an ANSI driver. This allows access to the Jet 3.5 databases and support of all existing ISAM file types.

When an ANSI application uses the ODBC Desktop Database Driver 4.0 and accesses Microsoft Access 4.0 or later, the driver exposes the data type as SQL_CHAR, SQL_VARCHAR, or SQL_LONGVARCHAR even though Jet 4.0 supports the wide version. Older versions of Jet do not support SQL_WCHAR, SQL_WVARCHAR, and SQL_WLONGVARCHAR. This restriction also applies in cases where the old formats are used with the Jet 4.0 Database Engine.

For more information concerning Unicode issues with ODBC, see Unicode in Programming Considerations.

Desktop Database Driver Performance Issues

To ensure compatibility with existing ANSI applications, the SQL_WCHAR, SQL_WVARCHAR, and SQL_WLONGVARCHAR data types are exposed as SQL_CHAR, SQL_VARCHAR, and SQL_LONGVARCHAR for Microsoft Access 4.0 or higher data sources. The data sources do not return WIDE CHAR data types but the data still must be sent to Jet in Wide Char form. It is important to understand that conversion will take place if a SQL_C_CHAR parameter or result column is bound to a SQL_CHAR data type in an ANSI application.

This conversion can be especially inefficient in terms of memory when a SQL_C_CHAR type is bound to a parameter of type LONGVARCHAR. Since the Jet 4.0 engine is unable to stream LONGTEXT parameter data, a UNICODE conversion buffer must be allocated that is twice the size of the SQL_C_CHAR ANSI buffer. The most efficient mechanism is for the application to perform the UNICODE conversion and bind the parameter as type SQL_C_WCHAR. When a parameter is marked as data-at-execution and the data is supplied in multiple calls to SQLPutData, a longtext data buffer is grown. One way to avoid the expense of growing this "Put Data" buffer is to supply an optional length via SQL_DATA_AT_EXEC_LEN(x), where x is the expected length of bytes. This will initialize the size of an internal PutData buffer to x bytes.

An efficient way to insert or update long data can be accomplished using SQLBulkOperations() or SQLSetPos() and setting the long data to SQL_DATA_AT_EXEC. (EXEC_LEN is ignored in this case.) Data can be streamed in chunks by calling SQLPutData multiple times, which will effectively append the data to the table.

When an application using a Jet 3.5 database through the Microsoft ODBC Desktop Database Drivers is upgraded to version 4.0, some performance degradation and an increased working set size may occur. This is because when a version 3.x database is opened using the new version 4.0 driver, it loads Jet 4.0. When Jet 4.0 opens the database and sees that the database is a 3.x version, it loads an Installable ISAM driver that is equivalent to loading the Jet 3.5 engine as well. To remove the performance and size penalty, the Jet 3.x database should be compacted into a Jet 4.0 format database. This will eliminate loading two Jet engines and minimize the code path to the data.

Also, the Jet 4.0 engine is a Unicode engine. All strings are stored and manipulated in Unicode. When an ANSI application accesses a Jet 3.x database through the Jet 4.0 engine, the data is converted from ANSI to Unicode and back to ANSI. If the database is updated to the version 4.0 format, the strings are converted to Unicode, removing one level of string conversion as well as minimizing the code path to the data by going through only one Jet engine.

Microsoft Access Driver Programming Considerations

This section contains the following topics.
Setting Options Programmatically for the Access Driver

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Size</td>
<td>The size of the internal buffer, in kilobytes, that is used by Microsoft Access to transfer data to and from the disk. The default buffer size is 2048 KB (displayed as 2048). Any integer value divisible by 256 can be entered.</td>
<td>To set this option dynamically, use the MAXBUFFERSIZE keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td>Data Source Name</td>
<td>A name that identifies the data source, such as Payroll or Personnel.</td>
<td>To set this option dynamically, use the DSN keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td>Database</td>
<td>A Microsoft Access data source can be set up without selecting or creating a database. If no database is provided upon setup, the user will be prompted to choose a database file when connecting to the data source.</td>
<td>To set this option dynamically, use the DBQ keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td>Description</td>
<td>An optional description of the data in the data source; for example, &quot;Hire date, salary history, and current review of all employees.&quot;</td>
<td>To set this option dynamically, use the DESCRIPTION keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td>Exclusive</td>
<td>If the Exclusive box is selected, the database will be opened in Exclusive mode and can be accessed by only one user at a time. Performance is enhanced when running in Exclusive mode.</td>
<td>To set this option dynamically, use the EXCLUSIVE keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td>ImplicitCommitSync</td>
<td>Determines how changes made outside of a transaction are written to the database. This value is initially set to &quot;Yes&quot;, which means that the Microsoft Access driver will wait for commits in an internal/implicit transaction to be completed.</td>
<td>This option is included in the Set Advanced Options dialog box for the Microsoft Access driver.</td>
</tr>
<tr>
<td>Page Timeout</td>
<td>Specifies the period of time, in milliseconds, that a page (if not used) remains in the buffer before being removed. For the Microsoft Access driver, the default is 500 milliseconds (0.5 seconds). This option applies to all data sources that use the ODBC driver. The page timeout cannot be 0 because of an inherent delay. The page timeout cannot be less than the inherent delay, even if the page timeout option is set below that value.</td>
<td>To set this option dynamically, use the PAGETIMEOUT keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td>Read Only</td>
<td>Designates the database as read-only.</td>
<td>To set this option dynamically, use the READONLY keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td>System Database</td>
<td>The full path of the Microsoft Access system database to be used with the Microsoft Access database you want to access. Click the System Database button to select the system database to be used. The ODBC Microsoft Access driver prompts the user for a name and password. The default name is Admin and the default password in Microsoft Access for the Admin user is an empty string. To increase the security of your Microsoft Access database, create a new user to replace the Admin user and delete the Admin user, or change the objects to which the Admin user has access.</td>
<td>To set this option dynamically, use the SYSTEMDB keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td>Threads</td>
<td>The number of background threads for the engine to use. For the Microsoft Access driver, this value defaults to 3, but can be changed. The user may want to increase the number of threads if there is a large amount of activity in the database. This option is included in the Set Advanced Options dialog box for the Microsoft Access driver.</td>
<td>To set this option dynamically, use the THREADS keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td>UserCommitSync</td>
<td>Determines whether the Microsoft Access driver will perform an explicit user-defined transactions asynchronously. This value is initially set to &quot;Yes&quot;, which means that the Microsoft Access driver will wait for commits in a user-defined transaction to be completed. Setting this option to False can have unpredictable consequences in a multi-user environment.</td>
<td>To set this option dynamically, use the USERCOMMITSYNC keyword in a call to SQLConfigDataSource.</td>
</tr>
</tbody>
</table>
# Microsoft Access Data Types

The following table shows the Microsoft Access data types, data types used to create tables, and ODBC SQL data types.

<table>
<thead>
<tr>
<th>Microsoft Access data type</th>
<th>Data type (CREATETABLE)</th>
<th>ODBC SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGBINARY[1]</td>
<td>LONGBINARY</td>
<td>SQL_LONGVARBINARY</td>
</tr>
<tr>
<td>BINARY</td>
<td>BINARY</td>
<td>SQL_BINARY</td>
</tr>
<tr>
<td>BIT</td>
<td>BIT</td>
<td>SQL_BIT</td>
</tr>
<tr>
<td>COUNTER</td>
<td>COUNTER</td>
<td>SQL_INTEGER</td>
</tr>
<tr>
<td>CURRENCY</td>
<td>CURRENCY</td>
<td>SQL_NUMERIC</td>
</tr>
<tr>
<td>DATE/TIME</td>
<td>DATETIME</td>
<td>SQL_TIMESTAMP</td>
</tr>
<tr>
<td>GUID</td>
<td>GUID</td>
<td>SQL_GUID</td>
</tr>
<tr>
<td>LONG BINARY</td>
<td>LONGBINARY</td>
<td>SQL_LONGVARBINARY</td>
</tr>
<tr>
<td>NUMBER (FieldSize= SINGLE)</td>
<td>SINGLE</td>
<td>SQL_REAL</td>
</tr>
<tr>
<td>NUMBER (FieldSize= DOUBLE)</td>
<td>DOUBLE</td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>NUMBER (FieldSize= BYTE)</td>
<td>UNSIGNED BYTE</td>
<td>SQL_TINYINT</td>
</tr>
<tr>
<td>NUMBER (FieldSize= INTEGER)</td>
<td>SHORT</td>
<td>SQL_SMALLINT</td>
</tr>
<tr>
<td>NUMBER (FieldSize= LONG INTEGER)</td>
<td>LONG</td>
<td>SQL_INTEGER</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC</td>
<td>SQL_NUMERIC</td>
</tr>
<tr>
<td>OLE</td>
<td>LONGBINARY</td>
<td>SQL_LONGVARBINARY</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>VARBINARY</td>
<td>SQL_VARBINARY</td>
</tr>
</tbody>
</table>

[1] Access 4.0 applications only. Maximum length of 4000 bytes. Behavior similar to LONGBINARY.

[2] ANSI applications only.

[3] Unicode and Access 4.0 applications only.

## Note

SQLGetTypeInfo returns ODBC data types. It will not return all Microsoft Access data types if more than one Microsoft Access type is mapped to the same ODBC SQL data type. All conversions in Appendix D of the ODBC Programmer's Reference are supported for the SQL data types listed in the previous table.

The following table shows limitations on Microsoft Access data types.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY, VARBINARY, and VARCHAR</td>
<td>Creating a BINARY, VARBINARY, or VARCHAR column of zero or unspecified length actually returns a 510-byte column.</td>
</tr>
<tr>
<td>BYTE</td>
<td>Even though a Microsoft Access NUMBER field with a FieldSize equal to BYTE is unsigned, a negative number can be inserted into the field when using the Microsoft Access driver.</td>
</tr>
<tr>
<td>CHAR, LONGVARCHAR,</td>
<td>A character string literal can contain any ANSI character (1-255 decimal). Use two consecutive single quotation marks (&quot;) to represent one single quotation mark (').</td>
</tr>
</tbody>
</table>
Procedures should be used to pass character data when using any special character in a character data type column.

**DATE**

Date values must be either delimited according to the ODBC canonical date format or delimited by the datetime delimiter (#). Otherwise, Microsoft Access will treat the value as an arithmetic expression and will not raise a warning or error.

For example, the date "March 5, 1996" must be represented as `{d 1996-03-05}` or #03/05/1996#; otherwise, if only 03/05/1993 is submitted, Microsoft Access will evaluate this as 3 divided by 5 divided by 1996. This value rounds up to the integer 0, and since the zero day maps to 1899-12-31, this is the date used.

A pipe character (|) cannot be used in a date value, even if enclosed in back quotes.

**GUID**

Data type limited to Microsoft Access 4.0.

**NUMERIC**

Data type limited to Microsoft Access 4.0.

More limitations on data types can be found in Data Type Limitations.

© 2016 Microsoft

**SQLGetInfo Returned Values for Access**

The following table lists the C-language #defines for the fInfoType argument and the corresponding values returned by SQLGetInfo. This information can be retrieved by passing the listed C-language #defines to SQLGetInfo in the fInfoType argument. For more information about the values returned by SQLGetInfo, see the ODBC Programmer’s Reference.

Note

Where SQLGetInfo returns a 32-bit bitmask, a vertical bar (|) represents a bitwise OR.

<table>
<thead>
<tr>
<th>InfoType</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESSIBLE_PROCEDURES</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_ACCESSIBLE_TABLES</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_ACTIVE_ENVIRONMENTS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_AGGREGATE_FUNCTIONS</td>
<td>All set</td>
</tr>
<tr>
<td>SQL_ALTER_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_ALTER_TABLE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_ASYNC_MODE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_ROW_COUNT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_SUPPORT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BOOKMARK_PERSISTENCE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CATALOG_LOCATION</td>
<td>SQL_QL_START</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME_SEPARATOR</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_TERM</td>
<td>&quot;Database&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_USAGE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_COLLATION_SEQ</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_COLUMN_ALIAS</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_CONVERT_BIGINT</td>
<td>SQL_CB_NON_NULL</td>
</tr>
<tr>
<td>SQL_CONVERT_BINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL Convert</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SQL_CONVERT_BIT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_CHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DATE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DECIMAL</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_DOUBLE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FLOAT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FUNCTIONS</td>
<td>SQL_FN_CVT_CONVERT</td>
</tr>
<tr>
<td>SQL_CONVERT_INTEGER</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_NUMERIC</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_REAL</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_SMALLINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TIME</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TIMESTAMP</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TINYINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_VARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_VARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CORRELATION_NAME</td>
<td>SQL_CN_ANY</td>
</tr>
<tr>
<td>SQL_CREATE_ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_TABLE</td>
<td>SQL_CT_CREATE_TABLE</td>
</tr>
<tr>
<td>SQL_CREATE_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_VIEW</td>
<td>SQL_CV_CREATE_VIEW</td>
</tr>
<tr>
<td>SQL_CURSOR_COMMIT_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSOR_ROLLBACK_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSOR_SENSITIVITY</td>
<td>SQL_UNSPECIFIED</td>
</tr>
<tr>
<td>SQL_DATA_SOURCE_NAME</td>
<td>The DSN from Odbc.ini, or &quot;&quot; if DRIVER keyword is used in Odbc.ini</td>
</tr>
<tr>
<td>SQL_DATA_SOURCE_READ_ONLY</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_DATABASE_NAME</td>
<td>File name</td>
</tr>
<tr>
<td>SQL_DATE_TIME_LITERALS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DBMS_NAME</td>
<td>&quot;ACCESS&quot;</td>
</tr>
<tr>
<td>SQL_DBMS_VER</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_DDL_INDEX</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_DEFAULT_TXN_ISOLATION</td>
<td>SQL_TXN_READ_COMMITTED</td>
</tr>
<tr>
<td>SQL Parameter</td>
<td>Value</td>
</tr>
<tr>
<td>---------------</td>
<td>-------</td>
</tr>
<tr>
<td>SQL_DESCRIBE_PARAMETER</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DRIVER_HDBC</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HENV</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HLIB</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HSTMT</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_NAME</td>
<td>&quot;OdbcJt32.dll&quot;</td>
</tr>
<tr>
<td>SQL_DRIVER_ODBC_VER</td>
<td>&quot;3.51.0000&quot;</td>
</tr>
<tr>
<td>SQL_DRIVER_VER</td>
<td>&quot;4.00.nn.nn (nnnn specifies the build date)&quot;</td>
</tr>
<tr>
<td>SQL_DROP_ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_TABLE</td>
<td>SQL_DT_DROP_TABLE</td>
</tr>
<tr>
<td>SQL_DROP_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_VIEW</td>
<td>SQL_DV_DROP_VIEW</td>
</tr>
<tr>
<td>SQL_EXPRESSIONS_IN_ORDERBY</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_FILE_USAGE</td>
<td>SQL_FILE_CATALOG</td>
</tr>
<tr>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1</td>
<td>SQL_CA1_NEXT</td>
</tr>
<tr>
<td>SQL_GETDATA_EXTENSIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_GROUP_BY</td>
<td>SQL_GB_GROUP_BY_CONTAINS_SELECT</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_CASE</td>
<td>SQL_IC_MIXED</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_QUOTE_CHAR</td>
<td>&quot;`&quot; (back quote)</td>
</tr>
<tr>
<td>SQL_KEYWORDS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_LIKE_ESCAPE_CLAUSE</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MAX_BINARY_LITERAL_LEN</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_CATALOG_NAME_LEN</td>
<td>66</td>
</tr>
<tr>
<td>SQL_MAX_CHAR_LITERAL_LEN</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_COLUMN_NAME_LEN</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_GROUP_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_INDEX</td>
<td>32</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_ORDER_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_SELECT</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_TABLE</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_CONCURRENT_ACTIVITIES</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_CURSOR_NAME_LEN</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_DRIVER_CONNECTIONS</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_INDEX_SIZE</td>
<td>255</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>SQL_MAX_PROCEDURE_NAME_LEN</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE</td>
<td>2096</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE_INCLUDES_LONG</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MAX_SCHEMA_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_STATEMENT_LEN</td>
<td>65000</td>
</tr>
<tr>
<td>SQL_MAX_TABLE_NAME_LEN</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_TABLES_IN_SELECT</td>
<td>16</td>
</tr>
<tr>
<td>SQL_MAX_USER_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MULT_RESULT_SETS</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MULTIPLE_ACTIVE_TXN</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_NEED_LONG_DATA_LEN</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_NON_NULLABLE_COLUMNS</td>
<td>SQL_NNC_NON_NULL</td>
</tr>
<tr>
<td>SQL_NULL_COLLATION</td>
<td>SQL_NC_LOW</td>
</tr>
<tr>
<td>SQL_NUMERIC_FUNCTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ODBC_SAG_CLI_CONFORMANCE</td>
<td>SQL_OSCC_COMPLIANT</td>
</tr>
<tr>
<td>SQL_ODBC_SQL_INTEGRITY</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_ODBC_VER</td>
<td>From Driver Manager</td>
</tr>
<tr>
<td>SQL_OJ_CAPABILITIES</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ORDER_BY_COLUMNS_IN_SELECT</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL OUTER_JOINS</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_PROCEDURE_TERM</td>
<td>&quot;QUERY&quot;</td>
</tr>
<tr>
<td>SQL_PROcedures</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_QUOTED_IDENTIFIER_CASE</td>
<td>SQL_IC_MIXED</td>
</tr>
<tr>
<td>SQL_ROW_UPDATES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_SCHEMA_TERM</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>SQL_SCHEMA_USAGE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_SCROLL_OPTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_SEARCH_PATTERN_ESCAPE</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_SERVER_NAME</td>
<td>&quot;ACCESS&quot;</td>
</tr>
<tr>
<td>SQL_SPECIAL_CHARACTERS</td>
<td>&quot;- @#$%^&amp;*-_+\n()&lt;&gt;,:;?/&gt;'[]&quot;</td>
</tr>
<tr>
<td>SQL_STRING_FUNCTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_SUBQUERIES</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_SYSTEM_FUNCTIONS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_TABLE_TERM</td>
<td>&quot;TABLE&quot;</td>
</tr>
<tr>
<td>SQL_TIMEDATE_ADD_INTERVALS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_TIMEDATE_DIFF_INTERVALS</td>
<td>0</td>
</tr>
</tbody>
</table>
### SQLColAttributes (Access Driver)

This topic provides Access Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_COLUMN_DISPLAY_SIZE</td>
<td>For LONGVARBINARY data, SQL_COLUMN_DISPLAY_SIZE is the maximum length of the column, not the maximum length of the column times 2.</td>
</tr>
<tr>
<td>SQL_OWNER_NAME</td>
<td>An empty string (&quot;&quot;) is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>SQL_QUALIFIER_NAME</td>
<td>The path to a database file is returned.</td>
</tr>
<tr>
<td>SQL_COLUMN_SEARCHABLE</td>
<td>LONGVARBINARY and LONGVARCHAR columns are reported as SQL_UNSEARCHABLE. Fixed-length and variable-length binary and character data types are searchable, even though LONGVARBINARY and LONGVARCHAR are not.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft
SQLColumns (Access Driver)

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a database file is returned.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>SQL_NO_NULLS is returned for columns that participate in a primary key or unique index.</td>
</tr>
</tbody>
</table>

SQLConfigDataSource (Access Driver)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLATINGSEQUENCE</td>
<td>The sequence in which the fields are sorted. This sets the same option as Collating Sequence in the setup dialog box.</td>
</tr>
<tr>
<td>COMPACT_DB</td>
<td>Performs data compaction on a database file. Has the following format: COMPACT_DB=&lt;path_name&gt;&lt;optional_sort_order&gt;&lt;optional_ENCRYPT keyword&gt;. When using the COMPACT_DB keyword in the same statement with a DSN keyword, this driver ignores the DSN keyword. Therefore, compacting a database and specifying a DSN is a two-step process.</td>
</tr>
<tr>
<td>CREATE_DB</td>
<td>Creates a database file. Has the following format: CREATE_DB=&lt;path_name&gt;&lt;optional_sort-order&gt;&lt;optional_ENCRYPT keyword&gt;, where the path name is the full path to a Microsoft Access database. An error will be returned if the path name specifies an existing database. The sort order will be as set up in the New Database dialog box displayed when the Create button is pressed in the Microsoft Access Setup dialog box. If no sort order is specified, General is used. When using the CREATE_DB keyword in the same statement with a DSN keyword, this driver ignores the DSN keyword. Therefore, creating a database and specifying a DSN is a two-step process. When using the CREATE_DB keyword, if the pathname of the Microsoft Access database to be created contains one or more spaces, then the entire pathname must be enclosed by double quotation marks, as shown in the following examples: &quot;C:\PROGRAM FILES\COMMON FILES\ MyAccess.mdb&quot; &quot;C:\PROGRAM FILES\Access2.mdb&quot; CREATE_DB=C:\TEMP\test.mdb (no quotation marks needed)</td>
</tr>
<tr>
<td>CREATE_SYSDB</td>
<td>Creates a system database file. Has the following format: CREATE_SYSDB=&lt;path-name&gt;&lt;optional-sort-order&gt;, where the path name is the full path to a Microsoft Access database. An error will be returned if the path name specifies an existing database. The sort order will be as set up in the New Database dialog box displayed when the Create button is clicked in the ODBC Microsoft Access Setup dialog box. If no sort order is specified, General is used.</td>
</tr>
</tbody>
</table>
| CREATE_V2DB      | Creates a database file that is compatible with Microsoft Access 2.0. Has the following format: CREATE_V2DB=<path-name><optional-sort-order>, where the path name is the full path to a Microsoft Access database. An error will be returned if the path name specifies an existing database. The sort order will be as set up in the New Database dialog box displayed when the Create button is pressed in the Microsoft Access Setup dialog box. If no sort order is specified, General is used. When using the CREATE_V2DB keyword in the same statement with a DSN keyword, this driver ignores the DSN keyword. Therefore, creating a database and specifying a DSN is a two-step process. When using the CREATE_V2DB keyword, if the pathname of the Microsoft Access database to be created contains one or more
spaces, then the entire pathname must be enclosed by double quotation marks, as shown in the following examples:

"C:\PROGRAM FILES\COMMON FILES\ MyAccess.mdb"
"C:\PROGRAM FILES\Access2.mdb"
CREATE_V2DB=C:\TEMP\test.mdb (no quotation marks needed)

<table>
<thead>
<tr>
<th><strong>DBQ</strong></th>
<th>The name of the database file. This sets the same option as <strong>Database</strong> in the setup dialog box.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DEFAULTDIR</strong></td>
<td>The path specification to the database file.</td>
</tr>
<tr>
<td><strong>DESCRIPTION</strong></td>
<td>A description of the data in the data source. This sets the same option as <strong>Description</strong> in the setup dialog box.</td>
</tr>
<tr>
<td><strong>DRIVER</strong></td>
<td>The path specification to the driver DLL.</td>
</tr>
<tr>
<td><strong>DRIVERID</strong></td>
<td>An integer ID for the driver. 25 (Microsoft Access)</td>
</tr>
<tr>
<td><strong>FIL</strong></td>
<td>File type MS Access for Microsoft Access</td>
</tr>
<tr>
<td><strong>IMPLICITCOMMITSYNC</strong></td>
<td>Determines whether the Microsoft Access driver will perform internal or implicit commits asynchronously. This value is initially set to &quot;Yes&quot;, which means that the Microsoft Access driver will wait for commits in an internal/implicit transaction to be completed. The value of this option should not be changed without careful consideration of the consequences. For more information about the option, see the <a href="#">Microsoft Jet Database Engine Programmer's Guide</a>. This sets the same option as <strong>ImplicitCommitSync</strong> in the setup dialog box.</td>
</tr>
<tr>
<td><strong>MAXBUFFERSIZE</strong></td>
<td>The size of the internal buffer, in kilobytes, that is used by Microsoft Access to transfer data to and from the disk. The default buffer size is 2048 KB (displayed as 2048). Any integer value divisible by 256 can be used. This sets the same option as <strong>Buffer Size</strong> in the setup dialog box.</td>
</tr>
<tr>
<td><strong>MAXSCANROWS</strong></td>
<td>The number of rows to be scanned when setting a column’s data type based upon existing data. A number from 1 to 16 can be entered for the rows to scan. The value defaults to 8; if it is set to 0, all rows are scanned. (A number outside the limit will return an error.) This sets the same option as <strong>Rows to Scan</strong> in the setup dialog box.</td>
</tr>
<tr>
<td><strong>PAGETIMEOUT</strong></td>
<td>Specifies the period of time, in milliseconds, that a page (if not used) remains in the buffer before being removed. The default is five-tenths of a second (0.5 seconds). Note that this option applies to all data sources that use the ODBC driver. This sets the same option as <strong>Page Timeout</strong> in the setup dialog box.</td>
</tr>
<tr>
<td><strong>PWD</strong></td>
<td>The password.</td>
</tr>
<tr>
<td><strong>READONLY</strong></td>
<td>TRUE to make file read-only; FALSE to make file not read-only. This sets the same option as <strong>Read Only</strong> in the setup dialog box.</td>
</tr>
<tr>
<td><strong>REPAIR_DB</strong></td>
<td>Repairs a database damaged by a failure that occurs during the commit process. When using the REPAIR_DB keyword in the same statement with a DSN keyword, this driver ignores the DSN keyword. Therefore, repairing a database and specifying a DSN is a two-step process.</td>
</tr>
<tr>
<td><strong>SYSTEMDB</strong></td>
<td>For the Microsoft Access driver, the path specification to the system database file. This sets the same option as <strong>System Database</strong> in the setup dialog box.</td>
</tr>
<tr>
<td><strong>THREADS</strong></td>
<td>The number of background threads for the engine to use. This value defaults to 3, but can be changed. This sets the same option as <strong>Threads</strong> in the setup dialog box.</td>
</tr>
<tr>
<td><strong>UID</strong></td>
<td>For the Microsoft Access driver, the user ID name used for login.</td>
</tr>
<tr>
<td><strong>USERCOMMITSYNC</strong></td>
<td>Determines whether the Microsoft Access driver will perform user-defined transactions asynchronously. This value is initially set to &quot;Yes&quot;, which means that the Microsoft Access driver will wait for commits in a user-defined transaction to be completed. The value of this option should not be changed without careful consideration of the consequences. For more information about the option, see the <a href="#">Microsoft Jet Database Engine Programmer's Guide</a>. This sets the same option as <strong>UserCommitSync</strong> in the setup dialog box.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft
SQLDriverConnect (Access Driver)

Note

This topic provides Access Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

SQLDriverConnect enables you to connect to a driver without creating a data source (DSN).

The following keywords are supported in the connection string for all drivers: DSN, DBQ, and FIL.

The UID and PWD keywords are also supported.

The PWD keyword should not include any of the special characters (see SQL_SPECIAL_CHARACTERS in SQLInfo Returned Values).

The following table shows the minimum keywords required to connect to each driver, and provides an example of keyword/value pairs used with SQLDriverConnect. For a full list of DRIVERID values, see SQLConfigDataSource.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Keywords Required</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Access</td>
<td>Driver, DBQ</td>
<td>Driver=(Microsoft Access Driver (*.mdb)); DBQ=c:\temp\sample.mdb</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

SQLGetInfo (Access Driver)

Note

This topic provides Access Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

SQLGetInfo supports the SQL_FILE_USAGE information type. The returned value is a 16-bit integer that indicates how the driver directly treats files in a data source:

- SQL_FILE_NOT_SUPPORTED — The driver is not a single-tier driver.
- SQL_FILE_TABLE — A single-tier driver treats files in a data source as tables.
- SQL_FILE_QUALIFIER — A single-tier driver treats files in a data source as a qualifier.

The ODBC driver returns SQL_FILE_QUALIFIER because each file is a complete database.

SQL_BOOKMARK_PERSISTENCE

SQL_BP_SCROLL | SQL_BP_UPDATE[1]

[1] Bookmarks persist after a commit but do not persist after a rollback.

SQL_CONVERT_BINARY

SQL_CVT_DOUBLE | SQL_CVT_FLOAT | SQL_CVT_INTEGER | SQL_CVT_NUMERIC | SQL_CVT_REAL | SQL_CVT_SMALLINT | SQL_CVT_VARCHAR | SQL_CVT_WVARCHAR

SQL_CONVERT_CHAR

SQL_CVT_DOUBLE | SQL_CVT_FLOAT | SQL_CVT_INTEGER | SQL_CVT_NUMERIC | SQL_CVT_REAL | SQL_CVT_SMALLINT | SQL_CVT_VARCHAR | SQL_CVT_WVARCHAR

SQL_CONVERT_DATE

SQL_CVT_DOUBLE | SQL_CVT_FLOAT | SQL_CVT_INTEGER | SQL_CVT_NUMERIC | SQL_CVT_REAL | SQL_CVT_SMALLINT | SQL_CVT_VARCHAR | SQL_CVT_WVARCHAR

SQL_CONVERT_VARCHAR

SQL(DBMS_VER)

<table>
<thead>
<tr>
<th>ISAM</th>
<th>Version</th>
<th>Format of version numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Access</td>
<td>2.0</td>
<td>02.00.0000</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>03.00.0000</td>
</tr>
<tr>
<td></td>
<td>3.5</td>
<td>03.50.0000</td>
</tr>
<tr>
<td></td>
<td>4.0</td>
<td>04.00.0000</td>
</tr>
</tbody>
</table>

Note

Versions 1.0 and 1.1 are not supported. Also, there is no difference in the data format in Microsoft Access versions 3.0, 7.0, and 97.

SQL_DDL_INDEX

SQL(DBL_CREATE_INDEX)

SQL(DBL_DROP_INDEX)

SQL_GETDATA_EXTENSIONS

SQL(GD_ANY_ORDER) | SQL(GD_ANY_COLUMN) | SQL(GD_BLOCK) | SQL(GD_BOUND)

SQL_KEYWORDS

ALPHANUMERIC
AUTOINCREMENT
BINARY
BOOLEAN
BYTE
COUNTER
DATABASE
DATATIME
DISALLOW
DISTINCTROW
DOUBLEFLOAT
FLOAT4
FLOAT8
GENERAL
IEEEDOUBLE
IEEE SINGLE
IGNORE
IMAGE
INTEGER1
INTEGER2
INTEGER4
LOGICAL
LOGICAL1
LONG
LONGBINARY
LONGCHAR
LONGTEXT
MEMO
MONEY
NOTE
NUMBER
OLEOBJECT
OWNERACCESS
PARAMETERS
PERCENT
PIVOT
SHORT
SINGLE
SINGLEFLOAT
STDEV
STDEVP
STRING
TABLEID
TEXT
TOP
TRANSFORM
UNsignedBYTE
VAR
VARBINARY
VARP
YESNO

SQL_NUMERIC_FUNCTIONS

SQL_FN_NUM_ABS | SQL_FN_NUM_ATAN | SQL_FN_NUM_CEILING | SQL_FN_NUM_COS | SQL_FN_NUM_EXP | SQL_FN_NUM_FLOOR | SQL_FN_NUM_LOG
| SQL_FN_NUM_MOD | SQL_FN_NUM_POWER | SQL_FN_NUM_RAND | SQL_FN_NUM_SIGN | SQL_FN_NUM_SIN | SQL_FN_NUM_SQRT | SQL_FN_NUM_TAN
SQL_OJ_CAPABILITIES
SQL_OJ_LEFT SQL_OJ_RIGHT SQL_OJ_NOT_ORDERED SQL_OJ_INNER SQL_OJ_ALL_COMPARISON_OPS

SQL_CATALOG_USAGE
SQL_QU_DML_STATEMENTS | SQL_QU_TABLE_DEFINITION | SQL_QU_INDEX_DEFINITION | SQL_QU_PROCEDURE_INVOCATION

SQL_SCROLL_OPTIONS
SQL_SO_FORWARD_ONLY | SQL_SO_STATIC | SQL_SO_KEYSET_DRIVEN

SQL_STRING_FUNCTIONS
SQL_FN_STR_ASCII | SQL_FN_STR_CHAR | SQL_FN_STR_CONCAT | SQL_FN_STR_LCASE | SQL_FN_STR_LEFT | SQL_FN_STR_LENGTH | SQL_FN_STR_LOCATE | SQL_FN_STR_LOCATE_2 | SQL_FN_STR_LTRIM | SQL_FN_STR_RIGHT | SQL_FN_STR_RTRIM | SQL_FN_STR_SPACE | SQL_FN_STR_SUBSTRING | SQL_FN_STR_UCASE

SQL_SUBQUERIES
SQL_SQ_COMPARISON | SQL_SQ_EXISTS | SQL_SQ_IN | SQL_SQ_QUANTIFIED | SQL_SQ_CORRELATED_SUBQUERIES

SQL_TIMEDATE_FUNCTIONS
SQL_FN_TD_CURDATE | SQL_FN_TD_CURTIME | SQL_FN_TD_DAYOFMONTH | SQL_FN_TD_DAYOFWEEK | SQL_FN_TD_DAYOFYEAR | SQL_FN_TD_HOUR | SQL_FN_TD_MINUTE | SQL_FN_TD_MONTH | SQL_FN_TD_NOW | SQL_FN_TD_SECOND | SQL_FN_TD_WEEK | SQL_FN_TD_YEAR

© 2016 Microsoft

**SQLGetTypeInfo (Access Driver)**

**Note**

This topic provides Access Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

The name of the type (TYPE_NAME) returned in the table produced by `SQLGetTypeInfo` will be the name most commonly used by the data source.

`SQL_ALL_EXCEPT_LIKE` will be returned in the SEARCHABLE column for the Byte, Counter, Double, Single, Long, and Short data types. (The LIKE capability can be achieved by converting the value to a character using the ODBC canonical conversion functions, then performing the comparison.)

© 2016 Microsoft

**SQLProcedureColumns (Access Driver)**

**Note**

This topic provides Access Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Application developers should look for driver-defined columns starting at the end of the result set and proceeding backward.

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMN_TYPE</td>
<td>SQL_PARAM_INPUT or SQL_RESULT_COL</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>This is a driver-specific column that is returned at the end of the result set. The SQL type of the column is an integer.</td>
</tr>
</tbody>
</table>
SQLSetConnectOption (Access Driver)

<table>
<thead>
<tr>
<th>fOption</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESS_MODE</td>
<td>The SQL_ACCESS_MODE fOption can be set to either SQL_MODE_READ_ONLY or SQL_MODE_READ_WRITE. However, the driver does not prevent updates if SQL_ACCESS_MODE is set to SQL_MODE_READ_ONLY.</td>
</tr>
<tr>
<td>SQL_AUTOCOMMIT</td>
<td>When the Microsoft Access driver is used, the SQL_AUTOCOMMIT option may be set to either SQL_AUTOCOMMIT_ON or SQL_AUTOCOMMIT_OFF, because the Microsoft Access driver supports transactions[1].</td>
</tr>
<tr>
<td>SQL_CURRENT_QUALIFIER</td>
<td>Supported.</td>
</tr>
<tr>
<td>SQL_LOGIN_TIMEOUT</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_OPT_TRACE</td>
<td>Supported.</td>
</tr>
<tr>
<td>SQL_OPT_TRACEFILE</td>
<td>Supported.</td>
</tr>
<tr>
<td>SQL_PACKET_SIZE</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_QUIET_MODE</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_TRANSLATE_DLL</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_TRANSLATION_OPTION</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_TXN_ISOLATION</td>
<td>SQL_TXN_ISOLATION is always SQL_TXN_READ_COMMITTED.</td>
</tr>
</tbody>
</table>

[1] Atomic transactions are not supported by the Microsoft Access driver. When committing a transaction using the Microsoft Access driver, a finite delay exists between the time the transaction is committed and the time the values are written to disk. This delay is determined by a delay inherent in the Microsoft Jet engine. The page timeout will not be less than a minimum value, even if the PageTimeout option is set below that value. As a result, there is no guarantee that committed data is stable, since changes may be made during the delay.

SQLStatistics (Access Driver)

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a database file is returned for Microsoft Access. Pattern matching is not supported in the szTableQualifier argument.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column, because owner name is not supported.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>Undelimited table name. Pattern matching is not supported in the szTableName argument.</td>
</tr>
<tr>
<td>INDEX_QUALIFIER</td>
<td>NULL is always returned.</td>
</tr>
<tr>
<td>INDEX_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Only SQL_TABLE_STAT or SQL_INDEX_OTHER will be returned for TYPE.</td>
</tr>
<tr>
<td>SEQ_IN_INDEX</td>
<td>Index-dependent.</td>
</tr>
</tbody>
</table>
SQLTables (Access Driver)

Note

This topic provides Access Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>szTableOwner</td>
<td>The only valid argument for szTableOwner is NULL because none of the drivers support owner names. With szTableOwner set to NULL, all tables are returned. NULL is returned in the TABLE_OWNER column.</td>
</tr>
<tr>
<td>szTableQualifier</td>
<td>In the TABLE_QUALIFIER column, SQLTables will return the path to a database file.</td>
</tr>
<tr>
<td>SzTableType</td>
<td>When the Microsoft Access driver is used, &quot;SYSTEM TABLE&quot; is supported for szTableType for system tables, &quot;SYNONYM&quot; is supported for attached tables, and &quot;VIEW&quot; is supported for row-returning queries.</td>
</tr>
</tbody>
</table>

See Also

SQLTables Function

SQLTransact (Access Driver)

Note

This topic provides Access Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

When the Microsoft Access driver is used, SQL_COMMIT and SQL_ROLLBACK are supported for the fType argument in a call to SQLTransact.

If a failure occurs during the commit process, the affected database can be repaired using the Repair Database option in the Microsoft Access driver setup, or through the use of the REPAIR_DB keyword in the SQLConfigDataSource function.

Encryption (Access Driver)

(This control is available only when Version 4.x format is selected.)

With the release of the 4.0 version of the Microsoft Jet Database Engine, database encryption is allowed when creating Access databases. See the Jet Database Engine Programmer’s Reference for additional information.
Using the ExtendedAnsiSQL Connection String Attribute (Access Driver)

Jet 4.0 has several new features that are not enabled by default in the ODBC Desktop Database Drivers version 4.0. To take advantage of these new features, an application will have to set a new connection attribute, add a new attribute to the connection string, or set the attribute in a DSN. The attribute is called ExtendedAnsiSQL and pertains only to the Microsoft ODBC Desktop Database Drivers 4.0 and later.

This section contains the following topics.

- Setting ExtendedAnsiSQL
- Enabling New Data Types by Setting ExtendedAnsiSQL
- Creating and Dropping Views and Procedures Using ExtendedAnsiSQL
- Data Truncation Detection Enabled Using ExtendedAnsiSQL
- Jet 4.0 Uses SQL-92 Reserved Words List when ExtendedAnsiSQL Set

© 2016 Microsoft

Setting ExtendedAnsiSQL

The attribute can be controlled in the connection string by adding the ExtendedAnsiSQL attribute:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtendedAnsiSQL=0</td>
<td>This setting does not enable the new features.</td>
</tr>
<tr>
<td>ExtendedAnsiSQL=1</td>
<td>This setting enables the new features.</td>
</tr>
</tbody>
</table>

The attribute can also be set in a DSN through the Advanced Options dialog box when configuring a DSN through Control Panel.

Setting the attribute to 0 disables the new features; setting it to 1 enables the new features.

The attribute can also be set using SQLSetConnectAttr(). The attribute value is 65501 and is set to a SQLINTEGER value of 1 or 0, as documented in the preceding table. It can be called before or after connecting, but it is better to call it after connecting because of the order in which the driver processes cached connection attributes and connection strings.

© 2016 Microsoft

Enabling New Data Types by Setting ExtendedAnsiSQL

Two new data types are available in Jet 4.0 databases when the ExtendedAnsiSQL flag is turned on: SQL_DECIMAL and SQL_NUMERIC. The default precision and scale are 18 and 0, respectively. Data accessed via ODBC that is typed as SQL_DECIMAL or SQL_NUMERIC will be mapped to Microsoft Jet Decimal instead of Currency.

When the ExtendedAnsiSQL flag is turned off, you cannot create tables with decimal or numeric types, and these types will not appear in SQLGetTypeInfo(). However, if the table contains the new data types, they can be used with the correct data types.

© 2016 Microsoft

Creating and Dropping Views and Procedures Using ExtendedAnsiSQL

When the ExtendedAnsiSQL flag is turned on, Jet 4.0 supports the ability to create and drop views and procedures through SQL. These views and procedures will not be visible through Access.
Data Truncation Detection Enabled Using ExtendedAnsiSQL

When the ExtendedAnsiSQL flag is turned on and the application is inserting data into a char or binary column and data is truncated, the truncation will be detected. When the ExtendedAnsiSQL flag is turned off, the data is truncated without warning, as it was in previous versions of the ODBC Desktop Database Drivers.

Jet 4.0 Uses SQL–92 Reserved Words List when ExtendedAnsiSQL_Set

When the ExtendedAnsiSQL flag is turned on, Jet 4.0 uses the SQL-92 reserved words list. Trying to use a SQL-92 reserved word as an unquoted object name will result in a syntax error. When the ExtendedAnsiSQL flag is turned off, the new reserved words can be used as object names as before.

Microsoft Excel Driver Programming Considerations

This section contains the following topics.

- Setting Options Programmatically for the Excel Driver
- Microsoft Excel Data Types
- SQLGetInfo Returned Values for Excel
- Other Excel Driver Programming Details

Setting Options Programmatically for the Excel Driver

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Source Name</td>
<td>A name that identifies the data source, such as Payroll or Personnel.</td>
<td>To set this option dynamically, use the DSN keyword in a call to SQLConfigDataSource.</td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td>A Microsoft Access data source can be set up without selecting or creating a database. If no database is provided upon setup, the user will be prompted to choose a database file when connecting to the data source.</td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>An optional description of the data in the data source; for example, &quot;Hire date, salary history, and current review of all employees.&quot;</td>
<td></td>
</tr>
<tr>
<td><strong>Directory</strong></td>
<td>Displays the currently selected directory. For Microsoft Excel 3.0/4.0 files, the path display is labeled &quot;Directory&quot;, while for Microsoft Excel 5.0, 7.0, or 97 files, the path display is labeled &quot;Workbook&quot;.</td>
<td></td>
</tr>
<tr>
<td><strong>Read Only</strong></td>
<td>Designates the database as read-only.</td>
<td></td>
</tr>
<tr>
<td><strong>Rows to Scan</strong></td>
<td>The number of rows to scan to determine the data type of each column. The data type is determined given the maximum number of kinds of data found. If data is encountered that does not match the data type guessed for the column, the data type will be returned as a NULL value. For the Microsoft Excel driver, you can enter a number from 1 to 16 for the rows to scan. The value defaults to 8; if it is set to 0, all rows are scanned. (A number outside the limit will return an error.)</td>
<td></td>
</tr>
<tr>
<td><strong>Select Directory</strong></td>
<td>Displays a dialog box where you can select a directory containing the files you want to access. When defining a data source directory (for all drivers except Microsoft Access), specify the directory where your most commonly used files are located. The ODBC driver uses this directory as the default directory. Copy other files into this directory if they are used frequently. Alternatively, you can qualify file names in a SELECT statement with the directory name: <code>SELECT * FROM C:\MYDIR\EMP</code> Or, you can specify a new default directory by using the <code>SQLSetConnectOption</code> function with the <code>SQL_CURRENT_QUALIFIER</code> option. For Microsoft Excel 3.0 or 4.0 files, the path display is labeled &quot;Directory&quot;, and the path selection button is labeled &quot;Select Directory&quot;. For Microsoft Excel 5.0, 7.0, or 97 files, the path display is labeled &quot;Workbook&quot;, and the path selection button is labeled &quot;Select Workbook&quot;. When defining a data source directory, specify the directory where your most commonly used Microsoft Excel files are located for Microsoft Excel 3.0/4.0, or the directory where the workbook file is located for Microsoft Excel 5.0, 7.0, or 97. <strong>Use Current Directory</strong> is disabled for Microsoft Excel 5.0, 7.0, and 97.</td>
<td></td>
</tr>
</tbody>
</table>

**Microsoft Excel Data Types**

The following table shows how Microsoft Excel driver data types are mapped to ODBC SQL data types. The Microsoft Excel driver assigns these data types to columns in Microsoft Excel tables based on the data in the column.

<table>
<thead>
<tr>
<th>Microsoft Excel data type</th>
<th>ODBC data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENCY</td>
<td>SQL_NUMERIC</td>
</tr>
<tr>
<td>DATETIME</td>
<td>SQL_TIMESTAMP</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>SQL_BIT</td>
</tr>
<tr>
<td>NUMBER</td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>TEXT</td>
<td>SQL_VARCHAR</td>
</tr>
</tbody>
</table>

**Note**

SQLGetTypeInfo returns ODBC SQL data types. All conversions in Appendix D of the ODBC Programmer's Reference are supported for the ODBC SQL data types listed earlier in this topic.
The following table shows limitations on Microsoft Excel data types.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypted data</td>
<td>The Microsoft Excel driver cannot read encrypted data.</td>
</tr>
<tr>
<td>Error Strings</td>
<td>The Microsoft Excel driver cannot return a character string for the Microsoft Excel error values (#N/A!, #VALUE!, #REF!, #DIV/0!, #NUM!, #NAME?, and #NULL!), but returns a NULL instead.</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>The value in a LOGICAL column is returned in a SQL_C_CHAR buffer as either 0 or 1.</td>
</tr>
<tr>
<td>NUMBER</td>
<td>If an integer column is created, numbers that are too big for the integer data type can be entered, and data containing non-integer values can be inserted, with the result that the column may be converted to SQL_DOUBLE.</td>
</tr>
<tr>
<td>TEXT</td>
<td>When the rows of a column contain more than one Microsoft Excel data type, the ODBC Microsoft Excel driver assigns the SQL_VARCHAR data type to the column. There is one exception to this: if the column contains only two or three of the datetime data types (DATE, TIME, and DATETIME), the ODBC Microsoft Excel driver assigns the SQL_TIMESTAMP data type to the column. Creating a TEXT column of zero or unspecified length actually returns a 255-byte column. A character string literal can contain any ANSI character (1-255 decimal). Use two consecutive single quotation marks (&quot;) to represent one single quotation mark ('). Inserting a NULL into a column with a data type other than SQL_VARCHAR will cause the data type of the column to change to SQL_VARCHAR.</td>
</tr>
</tbody>
</table>

More limitations on data types can be found in Data Type Limitations.

© 2016 Microsoft

SQLGetInfo Returned Values for Excel

The following table lists the C-language #defines for the fInfoType argument and the corresponding values returned by SQLGetInfo. This information can be retrieved by passing the listed C-language #defines to SQLGetInfo in the fInfoType argument. For more information about the values returned by SQLGetInfo, see the ODBC Programmer’s Reference.

<table>
<thead>
<tr>
<th>InfoType</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESSIBLE_PROCEDURES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_ACCESSIBLE_TABLES</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_ACTIVE_ENVIRONMENTS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_AGGREGATE_FUNCTIONS</td>
<td>All set</td>
</tr>
<tr>
<td>SQL_ALTER_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_ALTER_TABLE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_ASYNC_MODE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_ROW_COUNT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_SUPPORT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BOOKMARK_PERSISTENCE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CATALOG_LOCATION</td>
<td>SQL_QL_START</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME_SEPARATOR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CATALOG_TERM</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_VALUE</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SQL_CATALOG_USAGE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_COLLATION_SEQ</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_COLUMN_ALIAS</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_CONCAT_NULL_BEHAVIOR</td>
<td>SQL_CB_NON_NULL</td>
</tr>
<tr>
<td>SQL_CONVERT_BIGINT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_BINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_BIT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_CHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DATE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DECIMAL</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_DOUBLE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FLOAT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FUNCTIONS</td>
<td>SQL_FN_CVT_CONVERT</td>
</tr>
<tr>
<td>SQL_CONVERT_INTEGER</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_NUMERIC</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_REAL</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_SMALLINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TIME</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TIMESTAMP</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TINYINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_VARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_VARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CORRELATION_NAME</td>
<td>SQL_CN_ANY</td>
</tr>
<tr>
<td>SQL_CREATE_ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_TABLE</td>
<td>SQL_CT_CREATE_TABLE</td>
</tr>
<tr>
<td>SQL_CREATE_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_VIEW</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CURSOR_COMMIT_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSOR_ROLLBACK_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSOR_SENSITIVITY</td>
<td>SQL_UNSPECIFIED</td>
</tr>
<tr>
<td>SQL_DATA_SOURCE_NAME</td>
<td>The DSN from Odbc.ini, or &quot;&quot; if DRIVER keyword is used in Odbc.ini</td>
</tr>
<tr>
<td>SQL_DATA_SOURCE_READ_ONLY</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_DATABASE_NAME</td>
<td>Current database directory</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>SQL_DATETIME_LITERALS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DBMS_NAME</td>
<td>&quot;EXCEL&quot;</td>
</tr>
<tr>
<td>SQL_DBMS_VER</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_DDL_INDEX</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DEFAULT_TXN_ISOLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DESCRIBE_PARAMETER</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DRIVER_HDBC</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HENV</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HLIB</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HSTMT</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_NAME</td>
<td>&quot;OdbcJt32.dll&quot;</td>
</tr>
<tr>
<td>SQL_DRIVER_ODBC_VER</td>
<td>&quot;3.51.0000&quot;</td>
</tr>
<tr>
<td>SQL_DRIVER_VER</td>
<td>&quot;4.00.nnnn&quot; (nnnn specifies the build date)</td>
</tr>
<tr>
<td>SQL_DROP_ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_TABLE</td>
<td>SQL_DT_DROP_TABLE</td>
</tr>
<tr>
<td>SQL_DROP_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_VIEW</td>
<td>SQL_DV_DROP_VIEW</td>
</tr>
<tr>
<td>SQL_EXPRESSIONS_IN_ORDERBY</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_FILE_USAGE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1</td>
<td>SQL_CA1_NEXT</td>
</tr>
<tr>
<td>SQL_GETDATA_EXTENSIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_GROUP_BY</td>
<td>SQL_GB_GROUP_BY_CONTAINS_SELECT</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_CASE</td>
<td>SQL_IC_MIXED</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_QUOTE_CHAR</td>
<td>&quot;&quot; (back quote)</td>
</tr>
<tr>
<td>SQL_KEYWORDS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_LIKE_ESCAPE_CLAUSE</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MAX_BINARY_LITERAL_LEN</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_CATALOG_NAME_LEN</td>
<td>66</td>
</tr>
<tr>
<td>SQL_MAX_CHAR_LITERAL_LEN</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_MAX_COLUMN_NAME_LEN</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_GROUP_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_INDEX</td>
<td>0</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_ORDER_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_SELECT</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_TABLE</td>
<td>255</td>
</tr>
<tr>
<td>When using the Microsoft Excel Driver, a CREATE TABLE statement might allow 256 columns, but the 255-column limit is still valid and an insert into column 256 will fail.</td>
<td></td>
</tr>
<tr>
<td>SQL_MAX_CONCURRENT_ACTIVITIES</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_CURSOR_NAME_LEN</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_DRIVER_CONNECTIONS</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_INDEX_SIZE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_PROCEDURE_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE</td>
<td>65535</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE_INCLUDES_LONG</td>
<td>&quot;y&quot;</td>
</tr>
<tr>
<td>SQL_MAX_SCHEMA_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_STATEMENT_LEN</td>
<td>65000</td>
</tr>
<tr>
<td>SQL_MAX_TABLE_NAME_LEN</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_MAX_TABLES_IN_SELECT</td>
<td>16</td>
</tr>
<tr>
<td>SQL_MAX_USER_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MULT_RESULT_SETS</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MULTIPLE_ACTIVE_TXN</td>
<td>&quot;y&quot;</td>
</tr>
<tr>
<td>SQL_NEED_LONG_DATA_LEN</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_NON_NULLABLE_COLUMNS</td>
<td>SQL_NNC_NON_NULL</td>
</tr>
<tr>
<td>SQL_NULL_COLLATION</td>
<td>SQL_NC_LOW</td>
</tr>
<tr>
<td>SQL_NUMERIC_FUNCTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ODBC_SAG_CLI_CONFORMANCE</td>
<td>SQL_OSCC_COMPLIANT</td>
</tr>
<tr>
<td>SQL_ODBC_SQL_INTEGRITY</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_ODBC_VER</td>
<td>From Driver Manager</td>
</tr>
<tr>
<td>SQL_OJ_CAPABILITIES</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ORDER_BY_COLUMNS_IN_SELECT</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_OUTER_JOINS</td>
<td>&quot;y&quot;</td>
</tr>
<tr>
<td>SQL_PROCEDURE_TERM</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_PROCEDURES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_QUOTED_IDENTIFIER_CASE</td>
<td>SQL_IC_MIXED</td>
</tr>
<tr>
<td>SQL_ROW_UPDATES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_SCHEMA_TERM</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_SCHEMA_USAGE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_SCROLL_OPTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_SEARCH_PATTERN_ESCAPE</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_SERVER_NAME</td>
<td>&quot;EXCEL&quot;</td>
</tr>
</tbody>
</table>
### Other Excel Driver Programming Details

#### Note

This section provides Excel Driver-specific information about ODBC functions and other programming details. For general information about a function, see the appropriate topic under ODBC API Reference.

This section contains the following topics.

- SQLBindParameter (Excel Driver)
- SQLColAttributes (Excel Driver)
- SQLColumns (Excel Driver)
- SQLConfigDataSource (Excel Driver)
- SQLDriverConnect (Excel Driver)
- SQLGetInfo (Excel Driver)
- SQLGetTypeInfo (Excel Driver)
- SQLSetConnectOption (Excel Driver)
- SQLStatistics (Excel Driver)
- SQLTables (Excel Driver)
- SQLTransact (Excel Driver)
- Read-Only Status (Excel Driver)

#### SQLBindParameter (Excel Driver)

#### Note

This topic provides Excel Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

When the Microsoft Excel driver is used, executing an INSERT statement that uses a parameter to insert a NULL into a SQL_CHAR column will return SQL_SUCCESS_WITH_INFO with SQLSTATE 01004, "Data Truncated."
### SQLColAttributes (Excel Driver)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_COLUMN_DISPLAY_SIZE</td>
<td>For LONGVARBINARY data, SQL_COLUMN_DISPLAY_SIZE is the maximum length of the column, not the maximum length of the column times 2.</td>
</tr>
<tr>
<td>SQL_OWNER_NAME</td>
<td>An empty string (&quot;&quot;&quot;) is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>SQL_QUALIFIER_NAME</td>
<td>The path to a directory is returned.</td>
</tr>
<tr>
<td>SQL_COLUMN_SEARCHABLE</td>
<td>LONGVARBINARY and LONGVARCHAR columns are reported as SQL_UNSEARCHABLE. Fixed-length and variable-length binary and character data types are searchable, even though LONGVARBINARY and LONGVARCHAR are not.</td>
</tr>
</tbody>
</table>

**Note**

The above is not a complete list of the attributes returned by **SQLColAttributes**.

### SQLColumns (Excel Driver)

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a directory is returned.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>SQL_NO_NULLS is returned for columns that participate in a primary key or unique index.</td>
</tr>
</tbody>
</table>

### ODBC Jet SQLConfigDataSource (Excel Driver)

The **SQLConfigDataSource** function that is used to add, modify, or delete a data source dynamically uses the following keywords.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBQ</td>
<td>For the Microsoft Excel driver when accessing Microsoft Excel 5.0 or later files, the name of the workbook file. This sets the same option as Database in the setup dialog box.</td>
</tr>
<tr>
<td>DEFAULTDIR</td>
<td>The path specification to the directory. This sets the same option as Select Directory or Select Workbook in the setup dialog box.</td>
</tr>
</tbody>
</table>
DESCRIPTION
A description of the data in the data source.
This sets the same option as **Description** in the setup dialog box.

DRIVER
The path specification to the driver DLL.

DRIVERID
An integer ID for the driver.
- 534 (Microsoft Excel 3.0)
- 278 (Microsoft Excel 4.0)
- 22 (Microsoft Excel 5.0/7.0)
- 790 (Microsoft Excel 97-2003)

FIL
File type, for example, Excel 3.0, Excel 4.0, Excel 5.0, Excel 7.0, Excel 97, Excel 2000, or Excel 2003.

FIRSTROWHASNAMES
Indicates whether the cells of the first row of the range contain the column names for the table (1) or not (0).

MAXSCANROWS
The number of rows to be scanned when setting a column's data type based upon existing data.
A number from 1 to 16 can be entered for the rows to scan. The value defaults to 8; if it is set to 0, all rows are scanned. (A number outside the limit will return an error.)
This sets the same option as **Rows to Scan** in the setup dialog box.

READONLY
TRUE to make file read-only; FALSE to make file not read-only.
This sets the same option as **Read Only** in the setup dialog box.

THREADS
The number of background threads for the engine to use. For the Microsoft Access driver, this value defaults to 3, but can be changed. For the dBASE, MicrosoftExcelDriver this value is 3, and cannot be changed.
This sets the same option as **Threads** in the setup dialog box.

---

**SQLDriverConnect** (Excel Driver)

---

**Note**
This topic provides Excel Driver-specific information. For general information about this function, see the appropriate topic under **ODBC API Reference**.

The following keywords are supported in the connection string for all drivers: **DSN**, **DBQ**, and **FIL**.

The following table shows the minimum keywords required to connect to each driver, and provides an example of keyword/value pairs used with **SQLDriverConnect**. For a full list of DRIVERID values, see **SQLConfigDataSource**.

---

If DBQ or DefaultDir is not specified for the Microsoft Excel 3.0 or 4.0 driver, the driver will connect to the current directory.

---

<table>
<thead>
<tr>
<th>Driver</th>
<th>Keywords Required</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Excel 3.0 or 4.0</td>
<td>Driver, DriverID</td>
<td>Driver= (Microsoft Excel Driver (*.xls)); DBQ=c:\temp; DriverID=278</td>
</tr>
<tr>
<td>Microsoft Excel 5.0/7.0</td>
<td>Driver, DriverID, DBQ</td>
<td>Driver= (Microsoft Excel Driver (*.xls)); DBQ=c:\temp\sample.xls; DriverID=22</td>
</tr>
<tr>
<td>Microsoft Excel 97 and later</td>
<td>Driver, DriverID, DBQ</td>
<td>Driver= (Microsoft Excel Driver (*.xls)); DBQ=c:\temp\sample.xls; DriverID=790</td>
</tr>
</tbody>
</table>

---

**SQLGetInfo** (Excel Driver)
This topic provides Excel Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

SQLGetInfo supports the SQL_FILE_USAGE information type. The returned value is a 16-bit integer that indicates how the driver directly treats files in a data source:

- SQL_FILE_NOT_SUPPORTED — The driver is not a single-tier driver.
- SQL_FILE_TABLE — A single-tier driver treats files in a data source as tables.
- SQL_FILE_QUALIFIER — A single-tier driver treats files in a data source as a qualifier.

The ODBC driver returns SQL_FILE_TABLE for the Microsoft Excel driver because each file is a table.

### SQL_DBMS_VER

<table>
<thead>
<tr>
<th>ISAM</th>
<th>Version</th>
<th>Format of version numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Excel</td>
<td>3.0</td>
<td>03.00.0000</td>
</tr>
<tr>
<td></td>
<td>4.0</td>
<td>04.00.0000</td>
</tr>
<tr>
<td></td>
<td>5.0/7.0</td>
<td>05.00.0000</td>
</tr>
<tr>
<td></td>
<td>97/2000</td>
<td>08.00.0000</td>
</tr>
</tbody>
</table>

### SQL_FILE_USAGE

- SQL_FILE_TABLE (Excel 3.0/4.0)
- SQL_FILE_CATALOG (Excel 5.0/7.0)

### SQL_MAX_CHAR_LITERAL_LEN

- 255 (Excel 3.0/4.0/5.0/7.0)
- 65535 (Excel 97)

### SQL_MAX_COLUMN_NAME_LEN

- 30 (Excel 3.0/4.0)
- 64 (Excel 5.0/7.0/97)

### SQL_MAX_TABLE_NAME_LEN

- 12 (Excel 3.0/4.0)
- 31 (Excel 5.0/7.0/97)

### SQL CATALOG_NAME_SEPARATOR

- "\" (Excel 3.0/4.0)
- "." (Excel 5.0/7.0/97)

### SQL CATALOG_TERM

- "Directory" (Excel 3.0/4.0)
- "Workbook" (Excel 5.0/7.0/97)

### SQL CATALOG_USAGE
SQL_TIMEDATE_FUNCTIONS

SQL_FN_TD_CURDATE | SQL_FN_TD_CURTIME | SQL_FN_TD_DAYOFMONTH | SQL_FN_TD_DAYOFYEAR | SQL_FN_TD_HOUR | SQL_FN_TD_MINUTE | SQL_FN_TD_MONTH | SQL_FN_TD_NOW | SQL_FN_TD_SECOND | SQL_FN_TD_WEEK | SQL_FN_TD_YEAR

© 2016 Microsoft

SQLGetTypeInfo (Excel Driver)

Note

This topic provides Excel Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

The name of the type (TYPE_NAME) returned in the table produced by SQLGetTypeInfo will be the name most commonly used by the data source. SQL_ALL_EXCEPT_LIKE will be returned in the SEARCHABLE column for the Byte, Counter, Double, Single, Long, and Short data types. (The LIKE capability can be achieved by converting the value to a character using the ODBC canonical conversion functions, then performing the comparison.) When the Microsoft Excel driver is used, the ODBC type names are returned in the TYPE_NAME column that is returned by SQLGetTypeInfo.

© 2016 Microsoft

SQLSetConnectOption (Excel Driver)

Note

This topic provides Excel Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>fOption</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESS_MODE</td>
<td>The SQL_ACCESS_MODE fOption can be set to either SQL_MODE_READ_ONLY or SQL_MODE_READ_WRITE. However, the driver does not prevent updates if SQL_ACCESS_MODE is set to SQL_MODE_READ_ONLY.</td>
</tr>
<tr>
<td>SQL_AUTOCOMMIT</td>
<td>The Microsoft Excel driver only supports SQL_AUTOCOMMIT being set to ON (the default state), because they do not support transactions.</td>
</tr>
<tr>
<td>SQL_CURRENT_QUALIFIER</td>
<td>Supported.</td>
</tr>
<tr>
<td>SQL_LOGIN_TIMEOUT</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_OPT_TRACE</td>
<td>Supported.</td>
</tr>
<tr>
<td>SQL_OPT_TRACEFILE</td>
<td>Supported.</td>
</tr>
<tr>
<td>SQL_PACKET_SIZE</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_QUIET_MODE</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_TRANSLATE_DLL</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_TRANSLATION_OPTION</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_TXN_ISOLATION</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

SQLStatistics (Excel Driver)

Note
This topic provides Excel Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a directory. Pattern matching is not supported in the szTableQualifier argument.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>Undelimited table name. Pattern matching is not supported in the szTableName argument.</td>
</tr>
<tr>
<td>INDEX_QUALIFIER</td>
<td>NULL is always returned.</td>
</tr>
<tr>
<td>INDEX_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Only SQL_TABLE_STAT or SQL_INDEX_OTHER will be returned for TYPE.</td>
</tr>
<tr>
<td>SEQ_IN_INDEX</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLLATION</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>PAGES</td>
<td>NULL is always returned.</td>
</tr>
</tbody>
</table>

Filtering is based on uniqueness (the fUnique argument). The fAccuracy parameter is ignored.

© 2016 Microsoft

SQLTables (Excel Driver)

Note

This topic provides Excel Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>szTableOwner</td>
<td>The only valid argument for szTableOwner is NULL because none of the drivers support owner names. With szTableOwner set to NULL, all tables are returned. NULL is returned in the TABLE_OWNER column.</td>
</tr>
<tr>
<td>szTableQualifier</td>
<td>When the Microsoft Excel 3.0 or 4.0 driver is used, if you call SQLTables with a value for szTableQualifier that is not the name of an existing table, the driver will create a table with that name. In the TABLE_QUALIFIER column, SQLTables will return the path to a directory.</td>
</tr>
<tr>
<td>SzTableType</td>
<td>For Microsoft Excel 3.0 or 4.0, &quot;TABLE&quot; is the only table type supported. For later versions of Microsoft Excel files, &quot;SYSTEM TABLE&quot; is returned for sheet names (tables with a &quot;$&quot; on the end), and &quot;TABLE&quot; is returned for tables within worksheets.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

SQLTransact (Excel Driver)

Note

This topic provides Excel Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

This function supports SQL_COMMIT, but not SQL_ROLLBACK.
Read-Only Status (Excel Driver)

When the Microsoft Excel driver is used, data source tables are opened as read-only by default, and can be opened by only one user at a time. Even though tables have read-only status, however, applications can perform insertions and updates for Microsoft Excel tables.

When an application performs a Save As command on Microsoft Excel data through the Microsoft Excel driver, the application should create a new table and insert the data to be saved into the new table. Inserts result in an append to the table. No other operations can be performed on the table until it is closed and reopened. Once the table is closed, no subsequent insert can be performed because the table is then a read-only table.

It is possible to update values when using the Microsoft Excel driver, but a row cannot be deleted from a table based on a Microsoft Excel spreadsheet, so updates are not considered officially supported by the Microsoft Excel driver.

Paradox Driver Programming Considerations

This section contains the following topics:

- Setting Options Programmatically for the Paradox Driver
- Paradox Data Types
- SQLGetInfo Returned Values for Paradox
- Other Paradox Driver Programming Details

Note

Accessing Paradox ISAM files through the ODBC Desktop Database Drivers requires installation of the Borland database engine.

Setting Options Programmatically for the Paradox Driver

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory</td>
<td>Sets the targeted directory.</td>
<td>To set this option dynamically, use the <code>DEFAULTDIR</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Collating Sequence</td>
<td>The sequence in which the fields are sorted. The sequence can be ASCII (the default), International, Swedish-Finnish, or Norwegian-Danish.</td>
<td>To set this option dynamically, use the <code>COLLATINGSEQUENCE</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Description</td>
<td>An optional description of the data in the data source; for example, &quot;Hire date, salary history, and current review of all employees.&quot;</td>
<td>To set this option dynamically, use the <code>DESCRIPTION</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Exclusive</td>
<td>If the <code>Exclusive</code> box is selected, the database will be opened in Exclusive mode and can be accessed by only one user at a time. Performance is enhanced when running in Exclusive mode.</td>
<td>To set this option dynamically, use the <code>EXCLUSIVE</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Net Style</td>
<td>The network access style to use when accessing Paradox data: either &quot;3.x&quot; for Paradox 3.x or &quot;4.x&quot; for Paradox 4.x or 5.x. Can be set to &quot;3.x&quot; or &quot;4.x&quot; if the version is Paradox 4.x or 5.x; if the version is Paradox 3.x, the style must be &quot;3.x&quot;.</td>
<td>To set this option dynamically, use the <code>PARADOXNETSTYLE</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Page Timeout</td>
<td>Specifies the period of time, in tenths of a second, that a page (if not used) remains in the buffer before being removed. The default is 600 tenths of a second (60 seconds). This option applies to all data sources that use the ODBC driver. The page timeout cannot be 0 because of an inherent delay. The page timeout cannot be less than the inherent delay, even if the page timeout option is set below that value.</td>
<td>To set this option dynamically, use the <code>PAGETIMEOUT</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Read Only</td>
<td>Designates the database as read-only. To set this option dynamically, use the <code>READONLY</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Select Directory</td>
<td>Displays a dialog box where you can select a directory containing the files you want to access. When defining a data source directory specify the directory where your most commonly used files are located. The ODBC driver uses this directory as the default directory. Copy other files into this directory if they are used frequently. Alternatively, you can qualify file names in a SELECT statement with the directory name: <code>SELECT * FROM C:\MYDIR\EMP</code> Or, you can specify a new default directory by using the <code>SQLSetConnectOption</code> function with the <code>SQL_CURRENT_QUALIFIER</code> option. To set this option dynamically, use the <code>DEFAULTDIR</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
<td></td>
</tr>
<tr>
<td>Select Network Directory</td>
<td>The full path of the directory containing a Paradox lock database, because it contains either the Pdoxusr.net file (in Paradox 4.x) or the Paradox.net file (in Paradox 5.x). If the directory does not contain one of these files, the Paradox driver creates one. For information about these files, see the Paradox documentation. Before you can select a network directory, you must enter your Paradox user name in the User Name text box. Click <code>Select Network Directory</code> to select a network directory. To set this option dynamically, use the <code>PARADOXNETPATH</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
<td></td>
</tr>
<tr>
<td>User Name</td>
<td>The Paradox user name. This is the name displayed to other users of Paradox files when a lock is encountered. To set this option dynamically, use the <code>PARADOXUSERNAME</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
<td></td>
</tr>
</tbody>
</table>

**Paradox Data Types**

The ODBC Paradox driver maps Paradox data types to ODBC SQL data types. The following table lists all Paradox data types and shows the ODBC SQL data types they are mapped to.

<table>
<thead>
<tr>
<th>Paradox data type</th>
<th>ODBC data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHANUMERIC</td>
<td>SQL_VARCHAR</td>
</tr>
<tr>
<td>AUTOINCREMENT[1]</td>
<td>SQL_INTEGER</td>
</tr>
<tr>
<td>BCD[1]</td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>BYTES[1]</td>
<td>SQL_BINARY</td>
</tr>
<tr>
<td>DATE</td>
<td>SQL_DATE</td>
</tr>
<tr>
<td>IMAGE[2]</td>
<td>SQL_LONGVARBINARY</td>
</tr>
<tr>
<td>LOGICAL[1]</td>
<td>SQL_BIT</td>
</tr>
<tr>
<td>LONG[1]</td>
<td>SQL_INTEGER</td>
</tr>
<tr>
<td>MEMO[2]</td>
<td>SQL_LONGVARCHAR</td>
</tr>
<tr>
<td>MONEY[1]</td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>NUMBER</td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>SHORT</td>
<td>SQL_SMALLINT</td>
</tr>
<tr>
<td>TIME[1]</td>
<td>SQL_TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP[1]</td>
<td>SQL_TIMESTAMP</td>
</tr>
</tbody>
</table>

[1] Valid only for Paradox versions 5.x.

[2] Valid only for Paradox versions 4.x and 5.x.

Note
SQLGetTypeInfo returns ODBC SQL data types. All conversions in Appendix D of the ODBC Programmer’s Reference are supported for the ODBC SQL data types listed earlier in this topic.

The following table shows limitations on Paradox data types.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHANUMERIC</td>
<td>Creating an ALPHANUMERIC column of zero or unspecified length actually returns a 255-byte column.</td>
</tr>
<tr>
<td>BYTES</td>
<td>If you insert NULL into a binary column with the Paradox5 driver, it is changed to 0.</td>
</tr>
<tr>
<td>LONG</td>
<td>The maximum negative value supported by the Paradox driver for the Long data type in Paradox 5.x is not (-2^{31} (-2147483648)), as it should be since Long maps to the ODBC data type SQL_INTEGER. The maximum negative value supported for Long is actually (-2^{31} + 1 (-2147483647)).</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>When a value is inserted into a TIMESTAMP column by the Paradox driver, then subsequently retrieved from the column, the retrieved value may differ from the inserted value by as much as 1 second because of rounding.</td>
</tr>
</tbody>
</table>

More limitations on data types can be found in Data Type Limitations.

© 2016 Microsoft

### SQLGetInfo Returned Values for Paradox

The following table lists the C-language #defines for the fInfoType argument and the corresponding values returned by SQLGetInfo. This information can be retrieved by passing the listed C-language #defines to SQLGetInfo in the fInfoType argument. For more information about the values returned by SQLGetInfo, see the ODBC Programmer’s Reference.

**Note**

Where SQLGetInfo returns a 32-bit bitmask, a vertical bar (|) represents a bitwise OR.

<table>
<thead>
<tr>
<th>InfoType</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESSIBLE_PROCEDURES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_ACCESSIBLE_TABLES</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_ACTIVE_ENVIRONMENTS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_AGGREGATE_FUNCTIONS</td>
<td>All set</td>
</tr>
<tr>
<td>SQL_ALTER_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_ALTER_TABLE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ASYNC_MODE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_ROW_COUNT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_SUPPORT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BOOKMARK_PERSISTENCE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CATALOG_LOCATION</td>
<td>SQL_QL_START</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME_SEPARATOR</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_TERM</td>
<td>&quot;Directory&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_USAGE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_COLLATION_SEQ</td>
<td>**</td>
</tr>
<tr>
<td>SQL_COLUMN_ALIAS</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_CONCAT_NULL_BEHAVIOR</td>
<td>SQL_CB_NON_NULL</td>
</tr>
<tr>
<td>SQL_CONVERT_BIGINT</td>
<td>0</td>
</tr>
<tr>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>SQL_CONVERT_BINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_BIT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_CHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DATE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DECIMAL</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_DOUBLE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FLOAT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FUNCTIONS</td>
<td>SQL_FN_CVT_CONVERT</td>
</tr>
<tr>
<td>SQL_CONVERT_INTEGER</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_NUMERIC</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_REAL</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_SMALLINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TIME</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TIMESTAMP</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TINYINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_VARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_VARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CORRELATION_NAME</td>
<td>SQL_CN_ANY</td>
</tr>
<tr>
<td>SQL_CREATE_ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_TABLE</td>
<td>SQL_CT_CREATE_TABLE</td>
</tr>
<tr>
<td>SQL_CREATE_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_VIEW</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CURSOR_COMMIT_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSOR_ROLLBACK_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSOR_SENSITIVITY</td>
<td>SQL_UNSPECIFIED</td>
</tr>
<tr>
<td>SQL_DATA_SOURCE_NAME</td>
<td>The DSN from Odbc.ini, or &quot;&quot; if DRIVER keyword is used in Odbc.ini</td>
</tr>
<tr>
<td>SQL_DATA_SOURCE_READ_ONLY</td>
<td>&quot;N&quot; (This depends on the data source,)</td>
</tr>
<tr>
<td>SQL_DATABASE_NAME</td>
<td>Current database directory</td>
</tr>
<tr>
<td>SQL_DATETIME_LITERALS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DBMS_NAME</td>
<td>&quot;PARADOX&quot;</td>
</tr>
<tr>
<td>SQL_DBMS_VER</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_DDL_INDEX</td>
<td>Multiple values</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>SQL_DEFAULT_TXN_ISOLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DESCRIBE_PARAMETER</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DRIVER_HDBC</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HENV</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HLIB</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HSTMT</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_NAME</td>
<td>&quot;OdbcJt32.dll&quot;</td>
</tr>
<tr>
<td>SQL_DRIVER_ODBC_VER</td>
<td>&quot;3.51.0000&quot;</td>
</tr>
<tr>
<td>SQL_DRIVER_VER</td>
<td>&quot;4.00.nnnn&quot; (nnnn specifies the build date)</td>
</tr>
<tr>
<td>SQL_DROP_ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_TABLE</td>
<td>SQL_DT_DROP_TABLE</td>
</tr>
<tr>
<td>SQL_DROP_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_VIEW</td>
<td>SQL_DV_DROP_VIEW</td>
</tr>
<tr>
<td>SQL_EXPRESSIONS_IN_ORDERBY</td>
<td>&quot;y&quot;</td>
</tr>
<tr>
<td>SQL_FILE_USAGE</td>
<td>SQL_FILE_TABLE</td>
</tr>
<tr>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1</td>
<td>SQL_CA1_NEXT</td>
</tr>
<tr>
<td>SQL_GETDATA_EXTENSIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_GROUP_BY</td>
<td>SQL_GB_GROUP_BY_CONTAINS_SELECT</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_CASE</td>
<td>SQL_IC_MIXED</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_QUOTE_CHAR</td>
<td>&quot; &quot; (back quote)</td>
</tr>
<tr>
<td>SQL_KEYWORDS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_LIKE_ESCAPE_CLAUSE</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MAX_BINARY_LITERAL_LEN</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_CATALOG_NAME_LEN</td>
<td>66</td>
</tr>
<tr>
<td>SQL_MAX_CHAR_LITERAL_LEN</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_COLUMN_NAME_LEN</td>
<td>25</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_GROUP_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_INDEX</td>
<td>0 (Limit Unknown or Not Applicable)</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_ORDER_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_SELECT</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_TABLE</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_CONCURRENT_ACTIVITIES</td>
<td>0</td>
</tr>
<tr>
<td>SQL Parameter</td>
<td>Value</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>SQL_MAX_CURSOR_NAME_LEN</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_DRIVER_CONNECTIONS</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_INDEX_SIZE</td>
<td>1350</td>
</tr>
<tr>
<td>SQL_MAX_PROCEDURE_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE</td>
<td>1350</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE_INCLUDES_LONG</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MAX_SCHEMA_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_STATEMENT_LEN</td>
<td>65000</td>
</tr>
<tr>
<td>SQL_MAX_TABLE_NAME_LEN</td>
<td>12</td>
</tr>
<tr>
<td>SQL_MAX_TABLES_IN_SELECT</td>
<td>16</td>
</tr>
<tr>
<td>SQL_MAX_USER_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MULT_RESULT SETS</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MULTIPLE_ACTIVE_TXN</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_NEED_LONG_DATA_LEN</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_NON_NULLABLE_COLUMNS</td>
<td>SQL_NNC_NON_NULL</td>
</tr>
<tr>
<td>SQL_NULL_COLLATION</td>
<td>SQL_NC_LOW</td>
</tr>
<tr>
<td>SQL_NUMERIC_FUNCTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ODBC_SAG_CLI_CONFORMANCE</td>
<td>SQL_OSCC_COMPLIANT</td>
</tr>
<tr>
<td>SQL_ODBC_SQL_INTEGRITY</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_ODBC_VER</td>
<td>From Driver Manager</td>
</tr>
<tr>
<td>SQL_OJ_CAPABILITIES</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ORDER_BY_COLUMNS_IN_SELECT</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL Outer joins</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_PROCEDURE_TERM</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_PROCESURES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_QUOTED_IDENTIFIER_CASE</td>
<td>SQL_IC_MIXED</td>
</tr>
<tr>
<td>SQL_ROW_UPDATES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_SCHEMA_TERM</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_SCHEMA_USAGE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_SCROLL_OPTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_SEARCH_PATTERN_ESCAPE</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_SERVER_NAME</td>
<td>&quot;PARADOX&quot;</td>
</tr>
<tr>
<td>SQL_SPECIAL_CHARACTERS</td>
<td>&quot;~@#$%&amp;*-+-={(;?:/&gt;&lt;,!'[&quot;]&quot;</td>
</tr>
<tr>
<td>SQL_STRING_FUNCTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_SUBQUERIES</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_SYSTEM_FUNCTIONS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_TABLE_TERM</td>
<td>&quot;TABLE&quot;</td>
</tr>
<tr>
<td>SQL_TIMEDATE_ADD_INTERVALS</td>
<td>0</td>
</tr>
</tbody>
</table>
Other Paradox Driver Programming Details

Note
This section provides Paradox Driver-specific information about ODBC functions and other programming details. For general information about a function, see the appropriate topic under ODBC API Reference.

This section contains the following topics.

- SQLColAttributes (Paradox Driver)
- SQLColumns (Paradox Driver)
- SQLConfigDataSource (Paradox Driver)
- SQLDriverConnect (Paradox Driver)
- SQLGetInfo (Paradox Driver)
- SQLGetTypeInfo (Paradox Driver)
- SQLSetConnectOption (Paradox Driver)
- SQLStatistics (Paradox Driver)
- SQLTables (Paradox Driver)
- SQLTransact (Paradox Driver)
- CREATE INDEX for Paradox
- Multiple hstmts (Paradox Driver)

SQLColAttributes (Paradox Driver)

Note
This topic provides Paradox Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_COLUMN_DISPLAY_SIZE</td>
<td>For LONGVARBINARY data, SQL_COLUMN_DISPLAY_SIZE is the maximum length of the column, not the maximum length of the column times 2.</td>
</tr>
<tr>
<td>SQL_OWNER_NAME</td>
<td>An empty string (&quot;&quot;) is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>SQL_QUALIFIER_NAME</td>
<td>The path to a directory is returned.</td>
</tr>
<tr>
<td>SQL_COLUMN_SEARCHABLE</td>
<td>LONGVARBINARY and LONGVARCHAR columns are reported as SQL_UNSEARCHABLE.</td>
</tr>
<tr>
<td></td>
<td>Fixed-length and variable-length binary and character data types are searchable, even though LONGVARBINARY and LONGVARCHAR are not.</td>
</tr>
</tbody>
</table>
SQLColumns (Paradox Driver)

**Note**

The above is not a complete list of the attributes returned by SQLColAttributes.

### Table 1: SQLColumns (Paradox Driver)

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a directory is returned.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>SQL_NO_NULLS is returned for columns that participate in a primary key or unique index.</td>
</tr>
</tbody>
</table>

SQLConfigDataSource (Paradox Driver)

**Note**

This topic provides Paradox Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

The SQLConfigDataSource function that is used to add, modify, or delete a data source dynamically uses the following keywords.

### Table 2: SQLConfigDataSource (Paradox Driver)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLATINGSEQUENCE</td>
<td>The sequence in which the fields are sorted.</td>
</tr>
<tr>
<td></td>
<td>When the Paradox driver is used, the sequence can be ASCII (default), International, Swedish-Finnish, or Norwegian-Danish.</td>
</tr>
<tr>
<td></td>
<td>This sets the same option as Collating Sequence in the setup dialog box.</td>
</tr>
<tr>
<td>DBQ</td>
<td>The name of the database file.</td>
</tr>
<tr>
<td></td>
<td>This sets the same option as Database in the setup dialog box.</td>
</tr>
<tr>
<td>DEFAULTDIR</td>
<td>The path specification to the directory.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>A description of the data in the data source.</td>
</tr>
<tr>
<td></td>
<td>This sets the same option as Description in the setup dialog box.</td>
</tr>
<tr>
<td>DRIVER</td>
<td>The path specification to the driver DLL.</td>
</tr>
<tr>
<td>DRIVERID</td>
<td>An integer ID for the driver.</td>
</tr>
<tr>
<td></td>
<td>26 (Paradox 3.x)</td>
</tr>
<tr>
<td></td>
<td>282 (Paradox 4.x)</td>
</tr>
<tr>
<td></td>
<td>538 (Paradox 5.x)</td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td>Determines whether the database will be opened in exclusive mode (accessed by only one user at a time) or shared mode (accessed by more than one user at a time). Can be true (exclusive mode) or false (shared mode).</td>
</tr>
<tr>
<td></td>
<td>This sets the same option as Exclusive in the setup dialog box.</td>
</tr>
<tr>
<td>FIL</td>
<td>File type Paradox 3.x, Paradox 4.x, or Paradox 5.x</td>
</tr>
<tr>
<td>FILETYPE</td>
<td>File type for the Text driver (Text).</td>
</tr>
<tr>
<td>PAGETIMEOUT</td>
<td>Specifies the period of time, in tenths of a second, that a page (if not used) remains in the buffer before being removed. The default is 600 tenths of a second (60 seconds). Note that this option applies to all data sources that use the ODBC driver.</td>
</tr>
</tbody>
</table>
The following keywords are supported in the connection string for all drivers: `DSN`, `DBQ`, and `FIL`.

The `PWD` keyword is also supported. The PWD keyword should not include any of the special characters (see `SQL_SPECIAL_CHARACTERS` in `SQLGetInfo Returned Values`).

After a password-protected file has been opened by a user, other users are not allowed to open the same file.

The following table shows the minimum keywords required to connect to each driver, and provides an example of keyword/value pairs used with `SQLDriverConnect`. For a full list of DRIVERID values, see `SQLConfigDataSource`.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Keywords required</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradox</td>
<td>Driver, DriverID</td>
<td><code>Driver=(Microsoft Paradox Driver (*.db )); DBQ=c:\temp;DriverID=26</code></td>
</tr>
</tbody>
</table>

This topic provides Paradox Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

SQLGetInfo (Paradox Driver)

SQLGetInfo supports the `SQL_FILE_USAGE` information type. The returned value is a 16-bit integer that indicates how the driver directly treats files in a...
The ODBC driver returns SQL_FILE_TABLE because each file is a table.

**SQL_ALTER_TABLE**

**SQL_AT_ADD_COLUMN | SQL_AT_DROP_COLUMN**

**SQL_DDL_INDEX**

**SQL_DL_CREATE_INDEX | SQL_DL_DROP_INDEX**

**SQL_DBMS_VER**

<table>
<thead>
<tr>
<th>ISAM</th>
<th>Version</th>
<th>Format of version numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradox 3.x</td>
<td>03.00.0000</td>
<td></td>
</tr>
<tr>
<td>4.x</td>
<td>04.00.0000</td>
<td></td>
</tr>
<tr>
<td>5.x</td>
<td>05.00.0000</td>
<td></td>
</tr>
</tbody>
</table>

**SQL_CATALOG_USAGE**

**SQL_QU_DML_STATEMENTS | SQL_QU_TABLE_DEFINITION | SQL_QU_INDEX_DEFINITION**

**SQL_TIMEDATE_FUNCTIONS**

**SQL_FN_TD_DAYOFMONTH | SQL_FN_TD_DAYOFWEEK | SQL_FN_TD_DAYOFYEAR | SQL_FN_TD_HOUR | SQL_FN_TD_MINUTE | SQL_FN_TD_MONTH | SQL_FN_TD_SECOND | SQL_FN_TD_WEEK | SQL_FN_TD_YEAR**

---

© 2016 Microsoft

**SQLGetTypeInfo (Paradox Driver)**

---

This topic provides Paradox Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference. The name of the type (TYPE_NAME) returned in the table produced by SQLGetTypeInfo will be the name most commonly used by the data source. SQL_ALL_EXCEPT_LIKE will be returned in the SEARCHABLE column for the Byte, Counter, Double, Single, Long, and Short data types. (The LIKE capability can be achieved by converting the value to a character using the ODBC canonical conversion functions, and then performing the comparison.)

---

© 2016 Microsoft

**SQLSetConnectOption (Paradox Driver)**

---

Note
SQLStatistics (Paradox Driver)

### Column

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a directory. Pattern matching is not supported in the szTableQualifier argument.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>Undelimited table name. Pattern matching is not supported in the szTableName argument.</td>
</tr>
<tr>
<td>INDEX_QUALIFIER</td>
<td>NULL is always returned.</td>
</tr>
<tr>
<td>INDEX_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Only SQL_TABLE_STAT or SQL_INDEX_OTHER will be returned for TYPE.</td>
</tr>
<tr>
<td>SEQ_IN_INDEX</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLLATION</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>PAGES</td>
<td>NULL is always returned.</td>
</tr>
</tbody>
</table>

Filtering is based on uniqueness (the fUnique argument). The fAccuracy parameter is ignored.

© 2016 Microsoft

SQLTables (Paradox Driver)

### Column

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a directory. Pattern matching is not supported in the szTableQualifier argument.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>Undelimited table name. Pattern matching is not supported in the szTableName argument.</td>
</tr>
<tr>
<td>INDEX_QUALIFIER</td>
<td>NULL is always returned.</td>
</tr>
<tr>
<td>INDEX_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Only SQL_TABLE_STAT or SQL_INDEX_OTHER will be returned for TYPE.</td>
</tr>
<tr>
<td>SEQ_IN_INDEX</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLLATION</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>PAGES</td>
<td>NULL is always returned.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft
Note

This topic provides Paradox Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>szTableOwner</td>
<td>The only valid argument for szTableOwner is NULL because none of the drivers support owner names. With szTableOwner set to NULL, all tables are returned. NULL is returned in the TABLE_OWNER column.</td>
</tr>
<tr>
<td>szTableQualifier</td>
<td>In the TABLE_QUALIFIER column, SQLTables will return the path to a directory.</td>
</tr>
<tr>
<td>SzTableType</td>
<td>For Paradox files, “TABLE” is the only table type supported.</td>
</tr>
</tbody>
</table>

See Also

SQLTables Function

SQLTransact (Paradox Driver)

Note

This topic provides Paradox Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

This function supports SQL_COMMIT, but not SQL_ROLLBACK.

© 2016 Microsoft

CREATE INDEX for Paradox

The syntax of the CREATE INDEX statement for the ODBC Paradox driver is:

```
CREATE [ UNIQUE ] INDEX index-name
ON table-name
column-identifier [ASC]
[, column-identifier [ASC]...])
```

The ODBC Paradox driver does not support the DESC keyword in the ODBC SQL grammar for the CREATE INDEX statement. The table-name argument can specify the full path of the table.

If the keyword UNIQUE is specified, the ODBC Paradox driver will create a unique index. The first unique index is created as a primary index. This is a Paradox primary key file named table-name.PX. Primary indexes are subject to the following restrictions:

- The primary index must be created before any rows are added to the table.
- A primary index must be defined upon the first "n" columns in a table.
- Only one primary index is allowed per table.
- A table cannot be updated by the Paradox driver if a primary index is not defined on the table. (Note that this is not true for an empty table, which can be updated even if a unique index is not defined on the table.)
- The index-name argument for a primary index must be the same as the base name of the table, as required by Paradox.

If the keyword UNIQUE is omitted, the ODBC Paradox driver will create a non-unique index. This consists of two Paradox secondary index files named table-name.Xnn and table-name.Ynn, where nn is the number of the column in the table. Non-unique indexes are subject to the following restrictions:

- Before a non-unique index can be created for a table, a primary index must exist for that table.
For Paradox 3.x, the index-name argument for any index other than a primary index (unique or non-unique) must be the same as the column name. For Paradox 4.x and 5.x, the name of such an index can be, but doesn’t have to be, the same as the column name.

Only one column can be specified for a non-unique index.

Columns cannot be added once an index has been defined on a table. If the first column of the argument list of a CREATE TABLE statement creates an index, a second column cannot be included in the argument list.

For example, to use the sales order number and line number columns as the unique index on the SO_LINES table, use the statement:

```
CREATE UNIQUE INDEX SO_LINES
ON SO_LINES (SONum, LineNum)
```

To use the part number column as a non-unique index on the SO_LINES table, use the statement:

```
CREATE INDEX PartNum
ON SO_LINES (PartNum)
```

Note that when two CREATE INDEX statements are performed, the first statement will always create a primary index with the same name as the table and the second statement will always create a non-unique index with the same name as the column. These indexes will be named this way even if different names are entered in the CREATE INDEX statements and even if the index is labeled UNIQUE in the second CREATE INDEX statement.

**Note**

When you use the Paradox driver without implementing the Borland Database Engine, only read and append statements are allowed.

© 2016 Microsoft

**Multiple hstmts (Paradox Driver)**

When the ODBC Paradox driver is used, if you want to use more than one hstmt to execute queries on a table, the table must have a unique index (Paradox primary key).

© 2016 Microsoft

**dBASE Driver Programming Considerations**

This section contains the following topics.

- Setting Options Programmatically for the dBASE Driver
- dBASE Data Types
- SQLGetInfo Returned Values for dBASE
- Other dBASE Driver Programming Details

**Note**

Accessing dBASE ISAM files through the ODBC Desktop Database Drivers requires installation of the Borland database engine.

© 2016 Microsoft

**Setting Options Programmatically for the dBASE Driver**
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approximate Row Count</td>
<td>Determines whether table size statistics are approximated. This option applies to all data sources that use the ODBC driver.</td>
<td>To set this option dynamically, use the <code>STATISTICS</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Collating Sequence</td>
<td>The sequence in which the fields are sorted. The sequence can be: ASCII (the default) or International.</td>
<td>To set this option dynamically, use the <code>COLLATINGSEQUENCE</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Data Source Name</td>
<td>A name that identifies the data source, such as Payroll or Personnel.</td>
<td>To set this option dynamically, use the <code>DSN</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Database</td>
<td>A Microsoft Access data source can be set up without selecting or creating a database. If no database is provided upon setup, users will be prompted to select a database file when they connect to the data source.</td>
<td>To set this option dynamically, use the <code>DBQ</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Description</td>
<td>An optional description of the data in the data source; for example, &quot;Hire date, salary history, and current review of all employees.&quot;</td>
<td>To set this option dynamically, use the <code>DESCRIPTION</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Exclusive</td>
<td>If the Exclusive box is selected, the database will be opened in Exclusive mode and can be accessed by only one user at a time. Performance is enhanced when it runs in Exclusive mode.</td>
<td>To set this option dynamically, use the <code>EXCLUSIVE</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Page Timeout</td>
<td>Specifies the period of time, in tenths of a second, that a page (if not used) remains in the buffer before it is removed. The default is 600 tenths of a second (60 seconds). This option applies to all data sources that use the ODBC driver. The page timeout cannot be 0 because of an inherent delay. The page timeout cannot be less than the inherent delay, even if the page timeout option is set below that value.</td>
<td>To set this option dynamically, use the <code>PAGETIMEOUT</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Read Only</td>
<td>Designates the database as read-only.</td>
<td>To set this option dynamically, use the <code>READOONLY</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Select Directory</td>
<td>Displays a dialog box where you can select a directory that contains the files you want to access. When you define a data source directory, specify the directory where your most frequently used files are located. The ODBC driver uses this directory as the default directory. Copy other files into this directory if they are used frequently. Alternatively, you can qualify file names in a SELECT statement with the directory name: <code>SELECT * FROM C:\MYDIR\EMP</code> Or, you can specify a new default directory by using the <code>SQLSetConnectOption</code> function with the <code>SQL_CURRENT_QUALIFIER</code> option.</td>
<td>To set this option dynamically, use the <code>DEQUALUETDIR</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
<tr>
<td>Show Deleted Rows</td>
<td>Specifies whether rows that have been marked as deleted can be retrieved or positioned on. If cleared, deleted rows are not displayed; if selected, deleted rows are treated the same as non-deleted rows. The default is cleared.</td>
<td>To set this option dynamically, use the <code>DELETED</code> keyword in a call to <code>SQLConfigDataSource</code>.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**dBASE Data Types**

The following table shows how dBASE data types are mapped to ODBC SQL data types. Note that not all ODBC SQL data types are supported.

<table>
<thead>
<tr>
<th>dBASE data type</th>
<th>ODBC data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>SQL_VARCHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>SQL_DATE</td>
</tr>
<tr>
<td>FLOAT[1]</td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>SQL_BIT</td>
</tr>
<tr>
<td>MEMO</td>
<td>SQL_LONGVARCHAR</td>
</tr>
<tr>
<td>NUMERIC (BCD)</td>
<td>SQL_DOUBLE</td>
</tr>
</tbody>
</table>
Precision in dBASE III allows numbers with up to two-digit exponents and in dBASE IV numbers with up to three-digit exponents. Because numbers are stored as text, they are converted to numbers. If the number to convert does not fit in a field, unexplained results may occur.

While dBASE allows a precision and a scale to be specified with a NUMERIC data type, it is not supported by the ODBC dBASE driver. The ODBC dBASE driver always returns a precision of 15 and a scale of 0 for a NUMERIC data type.

A column created with the Numeric data type using the ODBC dBASE driver maps to the SQL_DOUBLE ODBC data type. Thus the data in this column is subject to rounding. This behavior is not the same as that of the NUMERIC data type in dBASE (type N), which is Binary Coded Decimal (BCD).

**Note**

`SQLGetTypeInfo` returns ODBC SQL data types. All conversions in Appendix D of the ODBC Programmer’s Reference are supported for the ODBC SQL data types listed earlier in this topic.

The following table shows limitations on dBASE data types.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>Creating a CHAR column of zero or unspecified length actually returns a 254-byte column.</td>
</tr>
<tr>
<td>Encrypted data</td>
<td>The dBASE driver does not support encrypted dBASE tables.</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>The dBASE driver cannot create an index on a LOGICAL column.</td>
</tr>
<tr>
<td>MEMO</td>
<td>The maximum length of a MEMO column is 65,500 bytes.</td>
</tr>
</tbody>
</table>

More limitations on data types can be found in Data Type Limitations.

© 2016 Microsoft

## SQLGetInfo Returned Values for dBASE

The following table lists the C-language #defines for the `fInfoType` argument and the corresponding values returned by `SQLGetInfo`. This information can be retrieved by passing the listed C-language #defines to `SQLGetInfo` in the `fInfoType` argument. For more information about the values returned by `SQLGetInfo`, see the ODBC Programmer’s Reference.

**Note**

Where `SQLGetInfo` returns a 32-bit bitmask, a vertical bar (|) represents a bitwise OR.

<table>
<thead>
<tr>
<th>InfoType</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESSIBLE_PROCEDURES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_ACCESSIBLE_TABLES</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_ACTIVE_ENVIRONMENTS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_AGREGATE_FUNCTIONS</td>
<td>All set</td>
</tr>
<tr>
<td>SQL_ALTER_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_ALTER_TABLE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ASYNC_MODE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_ROW_COUNT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_SUPPORT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BOOKMARK_PERSISTENCE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CATALOG_LOCATION</td>
<td>SQL_QL_START</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQLCatalogItem</td>
<td>Value</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME_SEPARATOR</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_TERM</td>
<td>&quot;Directory&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_USAGE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_COLLATION_SEQ</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_COLUMN_ALIAS</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_CONCAT_NULL_BEHAVIOR</td>
<td>SQL_CB_NON_NULL</td>
</tr>
<tr>
<td>SQL_CONVERT_BIGINT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_BINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_BIT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_CHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DATE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DECIMAL</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_DOUBLE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FLOAT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FUNCTIONS</td>
<td>SQL_FN_CVT_CONVERT</td>
</tr>
<tr>
<td>SQL_CONVERT_INTEGER</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_NUMERIC</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_REAL</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_SMALLINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TIME</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TIMESTAMP</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_TINYINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_VARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_VARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CORRELATION_NAME</td>
<td>SQL_CN_ANY</td>
</tr>
<tr>
<td>SQL_CREATE_ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_TABLE</td>
<td>SQL_CT_CREATE_TABLE</td>
</tr>
<tr>
<td>SQL_CREATE_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_VIEW</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CURSOR_COMMIT_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSOR_ROLLBACK_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSOR_SENSITIVITY</td>
<td>SQL_UNSPECIFIED</td>
</tr>
<tr>
<td>SQL_DATA_SOURCE_NAME</td>
<td>The DSN from Odbc.ini, or &quot;&quot; if DRIVER keyword is used in Odbc.ini</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>SQL_DATA_SOURCE_READ_ONLY</td>
<td>&quot;N&quot; (This depends on the data source.)</td>
</tr>
<tr>
<td>SQL_DATABASE_NAME</td>
<td>Current database directory</td>
</tr>
<tr>
<td>SQL_DATETIME_LITERALS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DBMS_NAME</td>
<td>&quot;DBASE&quot;</td>
</tr>
<tr>
<td>SQL_DBMS_VER</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_DDL_INDEX</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_DEFAULT_TXN_ISOLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DESCRIBE_PARAMETER</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DRIVER_HDBC</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HENV</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HLIB</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_HSTMT</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL_DRIVER_NAME</td>
<td>&quot;Odbc32.dll&quot;</td>
</tr>
<tr>
<td>SQL_DRIVER_ODBC_VER</td>
<td>&quot;3.51.0000&quot;</td>
</tr>
<tr>
<td>SQL_DRIVER_VER</td>
<td>&quot;4.00.1nnn&quot; (1nnn specifies the build date)</td>
</tr>
<tr>
<td>SQL_DROP_ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_TABLE</td>
<td>SQL_DT_DROP_TABLE</td>
</tr>
<tr>
<td>SQL_DROP_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_VIEW</td>
<td>SQL_DV_DROP_VIEW</td>
</tr>
<tr>
<td>SQL_EXPRESSIONS_IN_ORDERBY</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_FILE_USAGE</td>
<td>SQL_FILE_TABLE</td>
</tr>
<tr>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES</td>
<td>SQL_CA1_NEXT</td>
</tr>
<tr>
<td>SQL_GETDATA_EXTENSIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_GROUP_BY</td>
<td>SQL_GB_GROUP_BY_CONTAINS_SELECT</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_CASE</td>
<td>SQL_IC_UPPER (The qualifier is returned in mixed case so that Windows NT can locate the directory.)</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_QUOTE_CHAR</td>
<td>&quot; &quot; (back quote)</td>
</tr>
<tr>
<td>SQL_KEYWORDS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_LIKE_ESCAPE_CLAUSE</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MAX_BINARY_LITERAL_LEN</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_CATALOG_NAME_LEN</td>
<td>66</td>
</tr>
<tr>
<td>SQL_MAX_CHAR_LITERAL_LEN</td>
<td>254</td>
</tr>
<tr>
<td>SQL_MAX_COLUMN_NAME_LEN</td>
<td>10</td>
</tr>
<tr>
<td>SQL setting</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_GROUP_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_INDEX</td>
<td>0 (Limit Unknown or Not Applicable)</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_ORDER_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_SELECT</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_TABLE</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_CONCURRENT_ACTIVITIES</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_CURSOR_NAME_LEN</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_DRIVER_CONNECTIONS</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_INDEX_SIZE</td>
<td>220</td>
</tr>
<tr>
<td>SQL_MAX_PROCEDURE_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE</td>
<td>4000</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE_INCLUDES_LONG</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MAX_SCHEMA_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_STATEMENT_LEN</td>
<td>65000</td>
</tr>
<tr>
<td>SQL_MAX_TABLE_NAME_LEN</td>
<td>12</td>
</tr>
<tr>
<td>SQL_MAX_TABLES_IN_SELECT</td>
<td>16</td>
</tr>
<tr>
<td>SQL_MAX_USER_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MULT_RESULT_SETS</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MULTIPLE_ACTIVE_TXN</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_NEED_LONG_DATA_LEN</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_NON_NULLABLE_COLUMNS</td>
<td>SQL_NNC_NON_NULL</td>
</tr>
<tr>
<td>SQL_NULL_COLLATION</td>
<td>SQL_NC_LOW</td>
</tr>
<tr>
<td>SQL_NUMERIC_FUNCTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ODBC_SAG_CLI_CONFORMANCE</td>
<td>SQL_OSCC_COMPLIANT</td>
</tr>
<tr>
<td>SQL_ODBC_SQL_INTEGRITY</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_ODBC_VER</td>
<td>From Driver Manager</td>
</tr>
<tr>
<td>SQL_OJ_CAPABILITIES</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_ORDER_BY_COLUMNS_IN_SELECT</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_OUTER_JOINS</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_PROCEDURE_TERM</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_PROCEDURES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_QUOTED_IDENTIFIER_CASE</td>
<td>SQL_IC_MIXED</td>
</tr>
<tr>
<td>SQL_ROW_UPDATES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_SCHEMA_TERM</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_SCHEMA_USAGE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_SCROLL_OPTIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_SEARCH_PATTERN_ESCAPE</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>
Other dBASE Driver Programming Details

**Note**

This section provides dBASE Driver-specific information about ODBC functions and other programming details. For general information about a function, see the appropriate topic under ODBC API Reference.

This section contains the following topics:

- SQLColAttributes (dBASE Driver)
- SQLColumns (dBASE Driver)
- SQLConfigDataSource (dBASE Driver)
- SQLDriverConnect (dBASE Driver)
- SQLEndTran (dBASE Driver)
- SQLExecute (dBASE Driver)
- SQLEXception (dBASE Driver)
- SQLFetch (dBASE Driver)
- SQLGetTypeInfo (dBASE Driver)
- SQLGetTypeInfo (dBASE Driver)
- SQLSetConnOption (dBASE Driver)
- SQLStatistics (dBASE Driver)
- SQLTables (dBASE Driver)
- SQLTransact (dBASE Driver)
- dBASE Indexes (dBASE Driver)

**SQLColAttributes (dBASE Driver)**

**Note**

This topic provides dBASE Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_COLUMN_DISPLAY_SIZE</td>
<td>For LONGVARBINARY data, SQL_COLUMN_DISPLAY_SIZE is the maximum length of the column, not the maximum length</td>
</tr>
</tbody>
</table>
of the column times 2.

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_OWNER_NAME</td>
<td>An empty string (&quot;&quot;) is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>SQL_QUALIFIER_NAME</td>
<td>The path to a directory is returned.</td>
</tr>
<tr>
<td>SQL_COLUMN_SEARCHABLE</td>
<td>LONGVARBINARY and LONGVARCHAR columns are reported as SQL_UNSEARCHABLE. Fixed-length and</td>
</tr>
<tr>
<td></td>
<td>variable-length binary and character data types are searchable, even though LONGVARBINARY</td>
</tr>
<tr>
<td></td>
<td>and LONGVARCHAR are not.</td>
</tr>
</tbody>
</table>

Note

The above is not a complete list of the attributes returned by **SQLColAttributes**.

---

**SQLColumns (dBASE Driver)**

**Note**

This topic provides dBASE Driver-specific information. For general information about this function, see the appropriate topic under **ODBC API Reference**.

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a directory is returned.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>SQL_NO_NULLS is returned for columns that participate in a primary key or unique index.</td>
</tr>
</tbody>
</table>

---

**SQLConfigDataSource (dBASE Driver)**

**Note**

This topic provides dBASE Driver-specific information. For general information about this function, see the appropriate topic under **ODBC API Reference**.

The **SQLConfigDataSource** function that is used to add, modify, or delete a data source dynamically uses the following keywords.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLATINGSEQUENCE</td>
<td>The sequence in which the fields are sorted. The sequence can be: ASCII (the</td>
</tr>
<tr>
<td></td>
<td>default) or International. This sets the same option as Collating Sequence in</td>
</tr>
<tr>
<td></td>
<td>the setup dialog box.</td>
</tr>
<tr>
<td>DEFAULTDIR</td>
<td>The path specification to the directory.</td>
</tr>
<tr>
<td>DELETED</td>
<td>For the dBASE driver, specifies whether or not rows that have been marked as</td>
</tr>
<tr>
<td></td>
<td>deleted can be retrieved or positioned on. If set to 1, deleted rows are not</td>
</tr>
<tr>
<td></td>
<td>displayed; if set to 0, deleted rows are treated the same as non-deleted rows.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>A description of the data in the data source. This sets the same option as</td>
</tr>
<tr>
<td></td>
<td>Description in the setup dialog box.</td>
</tr>
<tr>
<td>DRIVER</td>
<td>The path specification to the driver DLL.</td>
</tr>
<tr>
<td>DRIVERID</td>
<td>An integer ID for the driver. 21 (dBASE III) 277 (dBASE IV)</td>
</tr>
</tbody>
</table>
SQLDriverConnect (dBASE Driver)

<table>
<thead>
<tr>
<th>File type dBase III, dBase IV, or dBase 5</th>
</tr>
</thead>
</table>

**PAGETIMEOUT**
Specifies the period of time, in tenths of a second, that a page (if not used) remains in the buffer before being removed. The default is 600 tenths of a second (60 seconds). Note that this option applies to all data sources that use the ODBC driver. This sets the same option as **Page Timeout** in the setup dialog box.

**READONLY**
TRUE to make file read-only; FALSE to make file not read-only. This sets the same option as **Read Only** in the setup dialog box.

**STATISTICS**
For the dBASE driver, determines whether table size statistics are approximated. Note that this option applies to all data sources that use the ODBC driver. This sets the same option as **Approximate Row Count** in the setup dialog box.

**THREADS**
The number of background threads for the engine to use. This value is 3 and cannot be changed. This sets the same option as **Threads** in the setup dialog box.

---

© 2016 Microsoft

SQLGetInfo (dBASE Driver)

**Note**
This topic provides dBASE Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

**SQLDriverConnect** enables you to connect to a driver without creating a data source (DSN).

The following keywords are supported in the connection string for all drivers: **DSN**, **DBQ**, and **FIL**.

When the Paradox driver is used, after a password-protected file has been opened by a user, other users are not allowed to open the same file.

The following table shows the minimum keywords required to connect to each driver, and provides an example of keyword/value pairs used with **SQLDriverConnect**. For a full list of DRIVERID values, see **SQLConfigDataSource**.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Keywords required</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dBASE</td>
<td>Driver, DriverID</td>
<td>Driver=(Microsoft dBASE Driver (*.dbf)); DBQ=c:\temp; DriverID=277</td>
</tr>
</tbody>
</table>

---

© 2016 Microsoft

SQLGetInfo supports the SQL_FILE_USAGE information type. The returned value is a 16-bit integer that indicates how the driver directly treats files in a data source:

- **SQL_FILE_NOT_SUPPORTED** — The driver is not a single-tier driver.
- **SQL_FILE_TABLE** — A single-tier driver treats files in a data source as tables.
- **SQL_FILE_QUALIFIER** — A single-tier driver treats files in a data source as a qualifier.

The ODBC driver returns SQL_FILE_TABLE because each file is a table.

**SQL_ALTER_TABLE**

**SQL_AT_ADD_COLUMN | SQL_AT_DROP_COLUMN**
### SQL_DBMS_VER

<table>
<thead>
<tr>
<th>ISAM</th>
<th>Version</th>
<th>Format of version numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBASE</td>
<td>3.0</td>
<td>03.00.0000</td>
</tr>
<tr>
<td></td>
<td>4.0</td>
<td>04.00.0000</td>
</tr>
<tr>
<td></td>
<td>5.0</td>
<td>05.00.0000</td>
</tr>
</tbody>
</table>

### SQL_DDL_INDEX

- SQL_DL_CREATE_INDEX
- SQL_DL_DROP_INDEX

### SQL_CATALOG_USAGE

- SQL_QU_DML_STATEMENTS
- SQL_QU_TABLE_DEFINITION
- SQL_QU_INDEX_DEFINITION

### SQL_TIMEDATE_FUNCTIONS

- SQL_FN_TD_DAYOFMONTH
- SQL_FN_TD_DAYOFWEEK
- SQL_FN_TD_DAYOFYEAR
- SQL_FN_TD_HOUR
- SQL_FN_TD_MINUTE
- SQL_FN_TD_MONTH
- SQL_FN_TD_SECOND
- SQL_FN_TD_WEEK
- SQL_FN_TD_YEAR

---

© 2016 Microsoft

### SQLGetTypeInfo (dBASE Driver)

**Note**

This topic provides dBASE Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

The name of the type (TYPE_NAME) returned in the table produced by `SQLGetTypeInfo` will be the name most commonly used by the data source.

`SQL_ALL_EXCEPT_LIKE` will be returned in the SEARCHABLE column for the Byte, Counter, Double, Single, Long, and Short data types. (The LIKE capability can be achieved by converting the value to a character using the ODBC canonical conversion functions, then performing the comparison.)

---

© 2016 Microsoft

### SQLSetConnectOption (dBASE Driver)

**Note**

This topic provides dBASE Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>fOption</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESS_MODE</td>
<td>The <code>SQL_ACCESS_MODE</code> fOption can be set to either SQL_MODE_READ_ONLY or SQL_MODE_READ_WRITE. However, the driver does not prevent updates if SQL_ACCESS_MODE is set to SQL_MODE_READ_ONLY.</td>
</tr>
<tr>
<td>SQL_AUTOCOMMIT</td>
<td>The dBASE driver only supports SQL_AUTOCOMMIT being set to ON (the default state), because it does not support transactions.</td>
</tr>
<tr>
<td>SQL_CURRENT_QUALIFIER</td>
<td>Supported.</td>
</tr>
<tr>
<td>Column</td>
<td>Comments</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a directory. Pattern matching is not supported in the szTableQualifier argument.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>Undelimited table name. Pattern matching is not supported in the szTableName argument.</td>
</tr>
<tr>
<td>INDEX_QUALIFIER</td>
<td>NULL is always returned.</td>
</tr>
<tr>
<td>INDEX_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Only SQL_TABLE_STAT or SQL_INDEX_OTHER will be returned for TYPE.</td>
</tr>
<tr>
<td>SEQ_IN_INDEX</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLLATION</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>PAGES</td>
<td>NULL is always returned.</td>
</tr>
</tbody>
</table>

Filtering is based on uniqueness (the fUnique argument). The fAccuracy parameter is ignored.

© 2016 Microsoft

SQLStatistics (dBASE Driver)

Note

This topic provides dBASE Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>szTableOwner</td>
<td>The only valid argument for szTableOwner is NULL because none of the drivers supports owner names. With szTableOwner set to NULL, all tables are returned. NULL is returned in the TABLE_OWNER column.</td>
</tr>
<tr>
<td>szTableQualifier</td>
<td>In the TABLE_QUALIFIER column, SQLTables will return the path to a directory.</td>
</tr>
</tbody>
</table>
For dBASE files, "TABLE" is the only table type supported.

SQLTransact (dBASE Driver)

Note

This topic provides dBASE Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference. This function supports SQL_COMMIT, but not SQL_ROLLBACK.

dBASE Indexes

The ODBC dBASE driver automatically opens and updates dBASE IV index files. You must use the Select Indexes dialog box displayed through the ODBC Data Source Administrator to associate dBASE III .ndx files with dBASE files.

The following limitations apply to the creation of dBASE indexes:

- All column names must be valid.
- All columns must be in the same ascending or descending order.
- The length of any single text column must be less than 100 bytes.
- If more than one column exists, all of the columns must be text columns and the sum of the column sizes must be less than 100 bytes.
- Memo fields cannot be indexed.
- An index must not be specified for the current set of fields (that is, duplicate indexes are not allowed).
- The index name must match the dBASE index naming convention. dBASE III requires that each index be in a separate file, each having an .ndx extension. In dBASE IV, indexes are created as tag names that are stored in a single .mdx file. The .mdx file has the same base name as the database file (for example, Emp.mdx is the index file for the Emp.dbf database).
- dBASE defines a unique index as one where only one record from a set with identical key values is added to the index.

Text File Driver Programming Considerations

This section contains the following topics:

- Setting Options Programmatically for the Text File Driver
- Text File Data Types
- SQLGetInfo Returned Values for Text Files
- Other Text File Driver Programming Details

Setting Options Programmatically for the Text File Driver

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Method</th>
</tr>
</thead>
</table>

© 2016 Microsoft
Data Source Name

A name that identifies the data source, such as Payroll or Personnel.

To set this option dynamically, use the DSN keyword in a call to SQLConfigDataSource.

Define Format

Displays the Define Text Format dialog box and enables you to specify the schema for individual tables in the data source directory.

This option cannot be set dynamically by a call to SQLConfigDataSource.

Description

An optional description of the data in the data source; for example, "Hire date, salary history, and current review of all employees."

To set this option dynamically, use the DESCRIPTION keyword in a call to SQLConfigDataSource.

Directory

Selects the targeted directory.

To set this option dynamically, use the DEFAULTDIR keyword in a call to SQLConfigDataSource.

Extensions List

Lists the file name extensions of the text files on the data source. When the Text driver is used, a file with no extension is created when the CREATE TABLE statement is executed with a name that has no extension. Other drivers create a file with a default extension when no extension is provided. To create a file with a .txt extension, the extension must be included in the name. To display files without extensions in the Define Text Format dialog box, "*.*" must be added to the Extensions List.

To set this option dynamically, use the EXTENSIONS keyword in a call to SQLConfigDataSource.

Read Only

Designates the database as read-only.

To set this option dynamically, use the READONLY keyword in a call to SQLConfigDataSource.

Rows to Scan

The number of rows to scan to determine the data type of each column. The data type is determined given the maximum number of kinds of data found. If data is encountered that does not match the data type guessed for the column, the data type will be returned as a NULL value.

For the Text driver, you may enter a number from 1 to 32767 for the number of rows to scan; however, the value will always default to 25. (A number outside the limit will return an error.)

To set this option dynamically, use the MAXSCANROWS keyword in a call to SQLConfigDataSource.

Select Directory

Displays a dialog box where you can select a directory containing the files you want to access.

When defining a data source directory specify the directory where your most commonly used files are located. The ODBC driver uses this directory as the default directory. Copy other files into this directory if they are used frequently. Alternatively, you can qualify file names in a SELECT statement with the directory name:

```
SELECT * FROM C:\MYDIR\EMP
```

Or, you can specify a new default directory by using the SQLSetConnectOption function with the SQL_CURRENT_QUALIFIER option.

To set this option dynamically, use the DEFAULTDIR keyword in a call to SQLConfigDataSource.

Text File Data Types

The following table shows how text data types are mapped to ODBC SQL data types. Note that not all ODBC SQL data types are supported by the ODBC Text driver.

<table>
<thead>
<tr>
<th>Text data type</th>
<th>ODBC data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>SQL_VARCHAR</td>
</tr>
<tr>
<td>DATETIME</td>
<td>SQL_TIMESTAMP</td>
</tr>
<tr>
<td>FLOAT</td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>SQL_INTEGER</td>
</tr>
</tbody>
</table>
Note

SQLGetTypeInfo returns ODBC data types. All conversions in Appendix D of the ODBC Programmer's Reference are supported for the SQL data types listed in the previous table.

The following table shows limitations on Text data types.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>Creating a CHAR column of zero or unspecified length actually returns a 255-bit column. In delimited files, a CHAR column may or may not have double quotation mark delimiters at the beginning and the end; in fixed-length files, double quotation marks are not used as delimiters.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>MM-DD-YY (for example, 01-17-92) MMM-DD-YY (for example, Jan-17-92) DD MMMM-YY (for example, 17-Jan-92) YYYY-MM-DD (for example, 1992-01-17) YYYY-MMM-DD (for example, 1992-Jan-17) Mixed date separators are not allowed within a table. The Text ISAM formats a DATETIME field in the United States or European format, depending upon the International setting in the Windows Control Panel.</td>
</tr>
<tr>
<td>FLOAT</td>
<td>The maximum width includes the sign and decimal point. In Schema.ini, the width is denoted as follows: 14.083 is FLOAT Width 6 -14.083 is FLOAT Width 7 +14.083 is FLOAT Width 7 14083. is FLOAT Width 6 ODBC always returns 8 for FLOAT columns. FLOAT columns can also be in scientific notation, for example: -3.04E+2 is Float Width 8 2SE4 is Float Width 4 Note Decimal and scientific notation cannot be mixed in a column. NULL values are represented by a blank padded string in fixed-length files, and are omitted in delimited files. Float data can be padded with leading blanks.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Valid values for INTEGER columns are 32767 to -32766. In Schema.ini, the width is denoted as follows: 14083 is INTEGER Width 5 0 is INTEGER Width 1 ODBC always returns 4 for INTEGER columns. The maximum width includes a sign. The maximum width of an INTEGER column is 11, although the width can be greater due to blanks that are allowed in fixed-format tables.</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>The theoretical limit on the width of a LONGCHAR column in either a fixed-length or delimited table is 65500K. The Text ISAM is more likely to provide reliable support up to about 32K.</td>
</tr>
</tbody>
</table>

More limitations on data types can be found in Data Type Limitations.
retrieved by passing the listed C-language #defines to SQLGetInfo in the fInfoType argument. For more information about the values returned by SQLGetInfo, see the ODBC Programmer’s Reference.

Note

Where SQLGetInfo returns a 32-bit bitmask, a vertical bar (|) represents a bitwise OR.

<table>
<thead>
<tr>
<th>InfoType</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESSIBLE_PROCEDURES</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_ACCESSIBLE_TABLES</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_ACTIVE_ENVIRONMENTS</td>
<td>0</td>
</tr>
<tr>
<td>SQL_AGGREGATE_FUNCTIONS</td>
<td>All set</td>
</tr>
<tr>
<td>SQL_ALTER_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_ALTER_TABLE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_ASYNC_MODE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_ROW_COUNT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BATCH_SUPPORT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_BOOKMARK_PERSISTENCE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CATALOG_LOCATION</td>
<td>SQL_QL_START</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_NAME_SEPARATOR</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_TERM</td>
<td>&quot;Directory&quot;</td>
</tr>
<tr>
<td>SQL_CATALOG_USAGE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_COLLATION_SEQ</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SQL_COLUMN_ALIAS</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_CONCAT_NULL_BEHAVIOR</td>
<td>SQL_CB_NON_NULL</td>
</tr>
<tr>
<td>SQL_CONVERT_BIGINT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_BINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_BIT</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_CHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DATE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_DECIMAL</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CONVERT_DOUBLE</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FLOAT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_FUNCTIONS</td>
<td>SQL_FN_CVT_CONVERT</td>
</tr>
<tr>
<td>SQL_CONVERT_INTEGER</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_LONGVARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_NUMERIC</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_REAL</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_CONVERT_SMALLINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL CONVERT TIME</td>
<td>Multiple values</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>SQL CONVERT_TIMESTAMP</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL CONVERT TINYINT</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL CONVERT VARBINARY</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL CONVERT VARCHAR</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL CORRELATION_NAME</td>
<td>SQL_CN_ANY</td>
</tr>
<tr>
<td>SQL_CREATE_ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CREATE_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL CREATE COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL CREATE DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL CREATE_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL CREATE TABLE</td>
<td>SQL_CT_CREATE_TABLE</td>
</tr>
<tr>
<td>SQL CREATE_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL CREATE VIEW</td>
<td>0</td>
</tr>
<tr>
<td>SQL_CURSOR_COMMIT_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSOR_ROLLBACK_BEHAVIOR</td>
<td>SQL_CB_CLOSE</td>
</tr>
<tr>
<td>SQL_CURSORSENSITIVITY</td>
<td>SQL_UNSPECIFIED</td>
</tr>
<tr>
<td>SQL DATA_SOURCE_NAME</td>
<td>The DSN from Odbc.ini, or &quot;&quot; if DRIVER keyword is used in Odbc.ini</td>
</tr>
<tr>
<td>SQL DATA_SOURCE_READ_ONLY</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL DATABASE_NAME</td>
<td>Current database directory</td>
</tr>
<tr>
<td>SQL DATETIME_LITERAL</td>
<td>0</td>
</tr>
<tr>
<td>SQL DBMS_NAME</td>
<td>&quot;TEXT&quot;</td>
</tr>
<tr>
<td>SQL DBMS_VER</td>
<td>ISAM: Text Version: 1.0 Format of version number: 01.00.0000</td>
</tr>
<tr>
<td>SQL DDL_INDEX</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DEFAULT_TXN_ISOLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL DESCRIBE_PARAMETER</td>
<td>0</td>
</tr>
<tr>
<td>SQL DRIVER_HDBC</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL DRIVER_HENV</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL DRIVER_HLIB</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL DRIVER_HSTMT</td>
<td>Handled by the Driver Manager.</td>
</tr>
<tr>
<td>SQL DRIVER_NAME</td>
<td>&quot;Odbcj32.dll&quot;</td>
</tr>
<tr>
<td>SQL DRIVER_ODBC_VER</td>
<td>&quot;3.51.0000&quot;</td>
</tr>
<tr>
<td>SQL DRIVER_VER</td>
<td>&quot;4.00.nnnn&quot; (nnnn specifies the build date)</td>
</tr>
<tr>
<td>SQL DROP ASSERTION</td>
<td>0</td>
</tr>
<tr>
<td>SQL DROP_CHARACTER_SET</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_COLLATION</td>
<td>0</td>
</tr>
<tr>
<td>------------------------</td>
<td>----</td>
</tr>
<tr>
<td>SQL_DROP_DOMAIN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_SCHEMA</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_TABLE</td>
<td>SQL_DT_DROP_TABLE</td>
</tr>
<tr>
<td>SQL_DROP_TRANSLATION</td>
<td>0</td>
</tr>
<tr>
<td>SQL_DROP_VIEW</td>
<td>SQLDV_DROP_VIEW</td>
</tr>
<tr>
<td>SQL_EXPRESSIONS_IN_ORDERBY</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_FILE_USAGE</td>
<td>SQL_FILE_TABLE</td>
</tr>
<tr>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1</td>
<td>SQL_CA1_NEXT</td>
</tr>
<tr>
<td>SQL_GETDATA_EXTENSIONS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_GROUP_BY</td>
<td>SQL_GB_GROUP_BY_CONTAINS_SELECT</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_CASE</td>
<td>SQL_IC_MIXED</td>
</tr>
<tr>
<td>SQL_IDENTIFIER_QUOTE_CHAR</td>
<td>&quot;`&quot; (back quote)</td>
</tr>
<tr>
<td>SQL_KEYWORDS</td>
<td>Multiple values</td>
</tr>
<tr>
<td>SQL_LIKE_ESCAPE_CLAUSE</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MAX_BINARY_LITERAL_LEN</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_CATALOG_NAME_LEN</td>
<td>66</td>
</tr>
<tr>
<td>SQL_MAX_CHAR_LITERAL_LEN</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_COLUMN_NAME_LEN</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_GROUP_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_INDEX</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_ORDER_BY</td>
<td>10</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_SELECT</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_COLUMNS_IN_TABLE</td>
<td>255</td>
</tr>
<tr>
<td>SQL_MAX_CONCURRENT_ACTIVITIES</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_CURSOR_NAME_LEN</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_DRIVER_CONNECTIONS</td>
<td>64</td>
</tr>
<tr>
<td>SQL_MAX_INDEX_SIZE</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_PROCEDURE_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE</td>
<td>65535</td>
</tr>
<tr>
<td>SQL_MAX_ROW_SIZE_INCLUDES_LONG</td>
<td>&quot;Y&quot;</td>
</tr>
<tr>
<td>SQL_MAX_SCHEMA_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MAX_STATEMENT_LEN</td>
<td>65000</td>
</tr>
<tr>
<td>SQL_MAX_TABLE_NAME_LEN</td>
<td>12</td>
</tr>
<tr>
<td>SQL_MAX_TABLES_IN_SELECT</td>
<td>16</td>
</tr>
<tr>
<td>SQL_MAX_USER_NAME_LEN</td>
<td>0</td>
</tr>
<tr>
<td>SQL_MULT_RESULT_SETS</td>
<td>&quot;N&quot;</td>
</tr>
<tr>
<td>SQL_MULTIPLE_ACTIVE_TXN</td>
<td>&quot;Y&quot;</td>
</tr>
</tbody>
</table>
Other Text File Driver Programming Details

Note

This section provides Text File Driver-specific information about ODBC functions and other programming details. For general information about a function, see the appropriate topic under ODBC API Reference.
This section contains the following topics:

- Defining Text Format (Text File Driver)
- Schema (Text File Driver)
- Schema.ini File (Text File Driver)
- Text File Format (Text File Driver)
- SQLColAttributes (Text File Driver)
- SQLColumns (Text File Driver)
- SQLConfigDataSource (Text File Driver)
- SQLDriverConnect (Text File Driver)
- SQLGetInfo (Text File Driver)
- SQLGetTypeInfo (Text File Driver)
- SQLSetConnectOption (Text File Driver)
- SQLStatistics (Text File Driver)
- SQLTables (Text File Driver)
- SQLTransact (Text File Driver)
- Creating and Opening Tables (Text File Driver)
- Read-Only Status (Text File Driver)

© 2016 Microsoft

Defining Text Format (Text File Driver)

When the Text driver is used, you can use the Define Text Format dialog box to define the format for columns in a selected file. This dialog box enables you to specify the schema for each data table. This information is written to a Schema.ini file in the data source directory. A separate Schema.ini file is created for each text data source directory.

**Note**

The same default file format applies to all new text data tables. All files created by the CREATE TABLE statement inherit those same default format values, which are set by selecting file format values in the Define Text Format dialog box with <default> chosen in the Tables list. The Text driver does not change the format of an existing text file to match the format defined in this dialog box, but returns an error when it uses the format, such as when it attempts to retrieve data from the text file.

The following options are available in the Define Text Format dialog box:

<table>
<thead>
<tr>
<th>Option</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Adds a column using the values in Data Type, Name, and Width from the dialog box, and if applicable, the Date Separator value from Schema.ini.</td>
</tr>
<tr>
<td>Characters</td>
<td>ANSI or OEM. OEM specifies a non-ANSI character set. This defaults to OEM if the format of the item selected in the Tables list has not been previously defined by this dialog box.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Indicates whether the columns of the first row of the selected table are to be used as column names. Either TRUE or FALSE. Defaults to FALSE if the format of the item selected in the Tables list has not been previously defined by this dialog box.</td>
</tr>
<tr>
<td>Headers</td>
<td>Lists the column names for each column in the selected table. The order of the columns reflects the order of the columns in the table. This list is enabled if a file has been selected in the Tables list.</td>
</tr>
<tr>
<td>Data Type</td>
<td>Can be BIT, BYTE, CHAR, CURRENCY, DATE, FLOAT, INTEGER, LONGCHAR, SHORT, or SINGLE. Date data types can be in the following formats: &quot;dd-mm-yy&quot;, &quot;mm-dd-yyyy&quot;, &quot;mm-dd-yyyy&quot;, &quot;yyyy-mm-dd&quot;, or &quot;yyyy-yyyy-mm-dd&quot;. &quot;mmm&quot; denotes numbers for months; &quot;mmm&quot; denotes letters for months.</td>
</tr>
<tr>
<td>Delimiter</td>
<td>Specifies the custom delimiter character to be used to separate columns. Enabled when the Custom Delimited format is selected. The delimiter can be only one character in length, and double quotation marks (&quot;) cannot be used as the delimiter character. (The delimiter cannot be specified in hexadecimal or decimal format.)</td>
</tr>
<tr>
<td>Format</td>
<td>Either delimited or fixed length. If delimited, indicates the type of delimiter used: comma (CSV), tab, or special character (custom). This defaults to CSV Delimited if the format of the item selected in the Tables list has not been previously defined by this dialog box.</td>
</tr>
</tbody>
</table>
defined by this dialog box.

If Format is fixed-length and Column Name Header is TRUE, the first line must be comma-delimited.

**Guess**

Automatically generates the column's data type, name, and width values for the columns in the selected table by scanning the table's contents according to the Format box selection. Enabled when the table format is delimited. Any previously defined columns in the Columns list are cleared and replaced with new entries. If Column Name Header is not selected, column names are generated automatically as "F1", "F2", and so on. No default value is shown in the Data Type box.

This functionality works only on columns that are less than 64,513 bytes.

**Modify**

Modifies the selected column using the values in Data Type, Name, and Width.

**Name**

Displays the name of the selected column. Can be used to specify a new column name for either an existing column or a new column.

If Column Name Header is TRUE, the column name displayed is ignored.

**Remove**

Deletes the selected column.

**Rows to Scan**

The number of rows that Setup or the driver will scan when setting the columns and column data types based upon existing data.

You can enter a number from 1 to 32767 for the number of rows to scan. This defaults to 25 if the format of the item selected in the Tables list has not been previously defined by this dialog box. (A number outside the limit will return an error.)

**Tables**

Contains a list of all files in the directory selected in the Text Setup dialog box that match the list of extensions specified.

When <default> is selected, and one of the following is true, the values of the table attributes in the Tables group are written to Schema.ini (no other entries in Schema.ini are touched):

- There is no Schema.ini in the specified directory.
- The Schema.ini file exists, but there is no section in Schema.ini for one of the Text files (with the specified extension) in the directory.
- The section for a Text file exists in Schema.ini, but the body is empty.

When <default> is selected, the Columns group is disabled.

**Width**

The width of the column can be changed for CHAR or LONGCHAR columns. The width defaults to 1 if the format of the item selected in the Tables list has not been previously defined by this dialog box.

For other data types, the width control is disabled and no value is displayed.

---

**Schema (Text File Driver)**

The schema includes information about each table (text file) in a data source, including the table's format, the number of rows to scan to determine column types, whether the first row of the table contains column names, whether the source file is written using an OEM or ANSI codepage, and each column's name, data type, and width.

---

**Schema.ini File (Text File Driver)**

When the Text driver is used, the format of the text file is determined by using a schema information file. The schema information file is always named Schema.ini and always kept in the same directory as the text data source. The schema information file provides the IISAM with information about the general format of the file, the column name and data type information, and several other data characteristics. A Schema.ini file is always required for accessing fixed-length data. You should use a Schema.ini file when your text table contains DateTime, Currency, or Decimal data, or any time that you want more control over the handling of the data in the table.
The Text ISAM will obtain initial values from the registry, not from Schema.ini. The same default file format applies to all new text data tables. All files that were created by the CREATE TABLE statement inherit those same default format values, which are set by selecting file format values in the Define Text Format dialog box with <default> chosen in the Tables list. If the values in the registry differ from the values in Schema.ini, the values in the registry will be overwritten by the values from Schema.ini.

Understanding Schema.ini Files

Schema.ini files provide schema information about the records in a text file. Each Schema.ini entry specifies one of five characteristics of the table:

- The text file name
- The file format
- The field names, widths, and types
- The character set
- Special data type conversions

The following sections discuss these characteristics.

Specifying the File Name

The first entry in Schema.ini is always the name of the text source file enclosed in square brackets. The following example illustrates the entry for the file Sample.txt:

```
[sample.txt]
```

Specifying the File Format

The Format option in Schema.ini specifies the format of the text file. The Text IISAM can read the format automatically from most character-delimited files. You can use any single character as a delimiter in the file except the double quotation mark ("). The Format setting in Schema.ini overrides the setting in the Windows Registry, file by file. The following table lists the valid values for the Format option.

<table>
<thead>
<tr>
<th>Format specifier</th>
<th>Table format</th>
<th>Schema.ini Format statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tab Delimited</td>
<td>Fields in the file are delimited by tabs.</td>
<td>Format=TabDelimited</td>
</tr>
<tr>
<td>CSV Delimited</td>
<td>Fields in the file are delimited by commas (comma-separated values).</td>
<td>Format=CSVDelimited</td>
</tr>
<tr>
<td>Custom Delimited</td>
<td>Fields in the file are delimited by any character you choose to input into the dialog box. All except the double quotation marks (&quot;) are allowed, including blank.</td>
<td>Format=Delimited(custom character) -or- With no delimiter specified: Format=Delimited( )</td>
</tr>
<tr>
<td>Fixed Length</td>
<td>Fields in the file are of a fixed length.</td>
<td>Format=FixedLength</td>
</tr>
</tbody>
</table>

Specifying the Fields

You can specify field names in a character-delimited text file in two ways:

- Include the field names in the first row of the table and set ColNameHeader to True.
- Specify each column by number and designate the column name and data type.

You must specify each column by number and designate the column name, data type, and width for fixed-length files.
The ColNameHeader setting in Schema.ini overrides the FirstRowHasNames setting in the Windows Registry, file by file.

The data types of the fields can also be determined. Use the MaxScanRows option to indicate how many rows should be scanned when determining the column types. If you set MaxScanRows to 0, the whole file is scanned. The MaxScanRows setting in Schema.ini overrides the setting in the Windows Registry, file by file.

The following entry indicates that Microsoft Jet should use the data in the first row of the table to determine field names and should examine the whole file to determine the data types used:

```
ColNameHeader=True
MaxScanRows=0
```

The next entry designates fields in a table by using the column number (Coln) option, which is optional for character-delimited files and required for fixed-length files. The example shows the Schema.ini entries for two fields, a 10-character CustomerNumber text field and a 30-character CustomerName text field:

```
Col1=CustomerNumber Text Width 10
Col2=CustomerName Text Width 30
```

The syntax of Coln is:

```
Coln=ColumnNametype [#]
```

Remarks

The following table describes each part of the Coln entry.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColumnName</td>
<td>The text name of the column. If the column name contains embedded spaces, you must enclose it in double quotation marks.</td>
</tr>
<tr>
<td>type</td>
<td>Data types are as follows:</td>
</tr>
<tr>
<td></td>
<td><strong>Microsoft Jet data types</strong></td>
</tr>
<tr>
<td></td>
<td>Bit</td>
</tr>
<tr>
<td></td>
<td>Byte</td>
</tr>
<tr>
<td></td>
<td>Short</td>
</tr>
<tr>
<td></td>
<td>Long</td>
</tr>
<tr>
<td></td>
<td>Currency</td>
</tr>
<tr>
<td></td>
<td>Single</td>
</tr>
<tr>
<td></td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>DateTime</td>
</tr>
<tr>
<td></td>
<td>Text</td>
</tr>
<tr>
<td></td>
<td>Memo</td>
</tr>
<tr>
<td></td>
<td><strong>ODBC data types</strong> Char (same as Text)</td>
</tr>
<tr>
<td></td>
<td>Float (same as Double)</td>
</tr>
<tr>
<td></td>
<td>Integer (same as Short)</td>
</tr>
<tr>
<td></td>
<td>LongChar (same as Memo)</td>
</tr>
<tr>
<td></td>
<td>Date date format</td>
</tr>
<tr>
<td>Width</td>
<td>The literal string value <em>width</em>. Indicates that the following number designates the width of the column (optional for character-delimited files; required for fixed-length files).</td>
</tr>
<tr>
<td>#</td>
<td>The integer value that designates the width of the column (required if <em>Width</em> is specified).</td>
</tr>
</tbody>
</table>
Selecting a Character Set

You can select from two character sets: ANSI and OEM. The CharacterSet setting in Schema.ini overrides the setting in the Windows Registry, file by file. The following example shows the Schema.ini entry that sets the character set to ANSI:

CharacterSet=ANSI

Specifying Data Type Formats and Conversions

The Schema.ini file contains several options that you can use to specify how data is converted or displayed. The following table lists each of these options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTimeFormat</td>
<td>Can be set to a format string that indicates dates and times. You should specify this entry if all date/time fields in the import/export are handled with the same format. All Microsoft Jet formats except A.M. and P.M. are supported. If there is no format string, the Windows Control Panel short date picture and time options are used.</td>
</tr>
<tr>
<td>DecimalSymbol</td>
<td>Can be set to any single character that is used to separate the integer from the fractional part of a number.</td>
</tr>
<tr>
<td>NumberDigits</td>
<td>Indicates the number of decimal digits in the fractional portion of a number.</td>
</tr>
<tr>
<td>NumberLeadingZeros</td>
<td>Specifies whether a decimal value less than 1 and more than –1 should contain leading zeros; this value can be either False (no leading zeros) or True.</td>
</tr>
<tr>
<td>CurrencySymbol</td>
<td>Indicates the currency symbol that can be used for currency values in the text file. Examples include the dollar sign ($) and Dm.</td>
</tr>
<tr>
<td>CurrencyPosFormat</td>
<td>Can be set to any of the following values:</td>
</tr>
<tr>
<td></td>
<td>• Currency symbol prefix with no separation ($)</td>
</tr>
<tr>
<td></td>
<td>• Currency symbol suffix with no separation ($)</td>
</tr>
<tr>
<td></td>
<td>• Currency symbol prefix with one character separation ($) ($)</td>
</tr>
<tr>
<td></td>
<td>• Currency symbol suffix with one character separation ($) ($)</td>
</tr>
<tr>
<td>CurrencyDigits</td>
<td>Specifies the number of digits used for the fractional part of a currency amount.</td>
</tr>
<tr>
<td>CurrencyNegFormat</td>
<td>Can be one of the following values:</td>
</tr>
<tr>
<td></td>
<td>• ($)</td>
</tr>
<tr>
<td></td>
<td>• –$</td>
</tr>
<tr>
<td></td>
<td>• $–</td>
</tr>
<tr>
<td></td>
<td>• $1–</td>
</tr>
<tr>
<td></td>
<td>• $1$</td>
</tr>
<tr>
<td></td>
<td>• $1–$</td>
</tr>
<tr>
<td></td>
<td>• –1 $</td>
</tr>
<tr>
<td></td>
<td>• –$1</td>
</tr>
<tr>
<td></td>
<td>• 1– $</td>
</tr>
<tr>
<td></td>
<td>• 1$</td>
</tr>
<tr>
<td></td>
<td>• 1$–</td>
</tr>
<tr>
<td></td>
<td>• –$</td>
</tr>
<tr>
<td></td>
<td>• –1 $</td>
</tr>
<tr>
<td></td>
<td>• 1 –$</td>
</tr>
<tr>
<td></td>
<td>• ($$)</td>
</tr>
</tbody>
</table>

This example shows the dollar sign, but you should replace it with the appropriate CurrencySymbol value in the actual program.

<table>
<thead>
<tr>
<th>CurrencyThousandSymbol</th>
<th>Indicates the single-character symbol that can be used for separating currency values in the text file by thousands.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrencyDecimalSymbol</td>
<td>Can be set to any single character that is used to separate the whole from the fractional part of a currency amount.</td>
</tr>
</tbody>
</table>
Text File Format (Text File Driver)

The ODBC Text driver supports both delimited and fixed-width text files. A text file consists of an optional header line and zero or more text lines.

Although the header line uses the same format as the other lines in the text file, the ODBC Text driver interprets the header line entries as column names, not data.

A delimited text line contains one or more data values separated by delimiters: commas, tabs, or a custom delimiter. The same delimiter must be used throughout the file. Null data values are denoted by two delimiters in a row with no data between them. Character strings in a delimited text line can be enclosed in double quotation marks ("""). No blanks can occur before or after delimited values.

The width of each data entry in a fixed-width text line is specified in a schema. Null data values are denoted by blanks.

Tables are limited to a maximum of 255 fields. Field names are limited to 64 characters, and field widths are limited to 32,766 characters. Records are limited to 65,000 bytes.

A text file can be opened only for a single user. Multiple users are not supported.

The following grammar, written for programmers, defines the format of a text file that can be read by the ODBC text driver:

<table>
<thead>
<tr>
<th>Format</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-italics</td>
<td>Characters that must be entered as shown</td>
</tr>
<tr>
<td>italics</td>
<td>Arguments that are defined elsewhere in the grammar</td>
</tr>
<tr>
<td>brackets ([])</td>
<td>Optional items</td>
</tr>
<tr>
<td>braces ({}</td>
<td>A list of mutually exclusive choices</td>
</tr>
<tr>
<td>vertical bars (</td>
<td>)</td>
</tr>
<tr>
<td>ellipses (...)</td>
<td>Items that can be repeated one or more times</td>
</tr>
</tbody>
</table>

The format of a text file:

```|
| text-file ::= |
| [delimited-header-line] [delimited-text-line]... end-of-file | |
| [fixed-width-header-line] [fixed-width-text-line]... end-of-file |
| delimited-header-line ::= delimited-text-line |
| delimited-text-line ::= blank-line |
| delimited-data [delimiter delimited-data]... end-of-line |
| fixed-width-header-line ::= fixed-width-text-line |
| fixed-width-text-line ::= blank-line |
| fixed-width-data [fixed-width-data]... end-of-line |
| end-of-file ::= <EOF> |
| blank-line ::= end-of-line |
| delimited-data ::= delimited-string | number | date | delimited-null |
| fixed-width-data ::= fixed-width-string | number | date | fixed-width-null |
```

The width of each column in a fixed-width text file is specified in the Schema.ini file.
The delimiter in a custom-delimited text file is specified in the Schema.ini file.

For delimited files, a NULL is represented by no data between two delimiters.

For fixed-width files, a NULL is represented by spaces.

SQLColAttributes (Text File Driver)

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_COLUMN_DISPLAY_SIZE</td>
<td>For LONGVARBINARY data, SQL_COLUMN_DISPLAY_SIZE is the maximum length of the column, not the maximum length of the column times 2.</td>
</tr>
<tr>
<td>SQL_OWNER_NAME</td>
<td>An empty string (&quot;&quot;&quot;) is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>SQL_QUALIFIER_NAME</td>
<td>The path to a directory is returned.</td>
</tr>
<tr>
<td>SQL_COLUMN_SEARCHABLE</td>
<td>LONGVARBINARY and LONGVARCHAR columns are reported as SQL_UNSEARCHABLE. Fixed-length and variable-length binary and character data types are searchable, even though LONGVARBINARY and LONGVARCHAR are not.</td>
</tr>
</tbody>
</table>

SQLColumns (Text File Driver)
### Note

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a directory is returned.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>SQL_NO_NULLS is returned for columns that participate in a primary key or unique index.</td>
</tr>
</tbody>
</table>

---

## SQLConfigDataSource (Text File Driver)

### Note

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

The **SQLConfigDataSource** function that is used to add, modify, or delete a data source dynamically uses the following keywords.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTERSET</td>
<td>For the Text driver, OEM or ANSI.</td>
</tr>
<tr>
<td>COLNAMEHEADER</td>
<td>For the Text driver, indicates whether the first record of data will specify the column names. Either TRUE or FALSE.</td>
</tr>
<tr>
<td>DEFAULTDIR</td>
<td>The path specification to the directory.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>A description of the data in the data source.</td>
</tr>
<tr>
<td></td>
<td>This sets the same option as Description in the setup dialog box.</td>
</tr>
<tr>
<td>DRIVER</td>
<td>The path specification to the driver DLL.</td>
</tr>
<tr>
<td>DRIVERID</td>
<td>An integer ID for the driver. 27 (Text)</td>
</tr>
<tr>
<td>EXTENSIONS</td>
<td>Lists the file name extensions of the Text files on the data source.</td>
</tr>
<tr>
<td></td>
<td>This sets the same option as Extensions List in the setup dialog box.</td>
</tr>
<tr>
<td>FIL</td>
<td>File type Text</td>
</tr>
<tr>
<td>FILETYPE</td>
<td>File type for the Text driver (Text).</td>
</tr>
<tr>
<td>FORMAT</td>
<td>For the Text driver, can be FIXEDLENGTH, TABDELIMITED, CSVDELIMITED (by a comma), or DELIMITED() (by the special character specified in the parentheses). The special character is one character in length and can be in character, decimal, or hexadecimal format.</td>
</tr>
<tr>
<td>MAXSCANROWS</td>
<td>The number of rows to be scanned when setting a column’s data type based upon existing data.</td>
</tr>
<tr>
<td></td>
<td>For the Text driver, you can enter a number from 1 to 32767 for the number of rows to scan; however, the value will always default to 25. (A number outside the limit will return an error.)</td>
</tr>
<tr>
<td></td>
<td>This sets the same option as Rows to Scan in the setup dialog box.</td>
</tr>
<tr>
<td>READONLY</td>
<td>TRUE to make file read-only; FALSE to make file not read-only.</td>
</tr>
<tr>
<td></td>
<td>This sets the same option as Read Only in the setup dialog box.</td>
</tr>
</tbody>
</table>

---

## SQLDriverConnect (Text File Driver)

### Note

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.
SQLDriverConnect enables you to connect to a driver without creating a data source (DSN).

The following keywords are supported in the connection string for all drivers: DSN, DBQ, and FIL.

The following table shows the minimum keywords required to connect to each driver, and provides an example of keyword/value pairs used with SQLDriverConnect. For a full list of DRIVERID values, see SQLConfigDataSource.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Keywords required</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Driver</td>
<td>Driver=(Microsoft Text Driver (<em>.txt;</em>.csv)); DefaultDir=c:\temp</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**SQLGetInfo (Text File Driver)**

**Note**

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

SQLGetInfo supports the SQL_FILE_USAGE information type. The returned value is a 16-bit integer that indicates how the driver directly treats files in a data source:

- SQL_FILE_NOT_SUPPORTED — The driver is not a single-tier driver.
- SQL_FILE_TABLE — A single-tier driver treats files in a data source as tables.
- SQL_FILE_QUALIFIER — A single-tier driver treats files in a data source as a qualifier.

The ODBC driver returns SQL_FILE_TABLE for the Text driver, because each file is a table.

**SQL_DBMS_VER**

<table>
<thead>
<tr>
<th>ISAM</th>
<th>Version</th>
<th>Format of version numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>1.0</td>
<td>01.00.0000</td>
</tr>
</tbody>
</table>

**SQL_CATALOG_USAGE**

SQL_QU_DML_STATEMENTS | SQL_QU_TABLE_DEFINITION

**SQL_TIMEDATE_FUNCTIONS**

SQL_FN_TD_CURDATE | SQL_FN_TD_CURTIME | SQL_FN_TD_DAYOFMONTH | SQL_FN_TD_DAYOFWEEK | SQL_FN_TD_DAYOFYEAR | SQL_FN_TD_HOUR | SQL_FN_TD_MINUTE | SQL_FN_TD_MONTH | SQL_FN_TD_NOW | SQL_FN_TD_SECOND | SQL_FN_TD_WEEK | SQL_FN_TD_YEAR

© 2016 Microsoft

**SQLGetTypeInfo (Text File Driver)**

**Note**

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

The name of the type (TYPE_NAME) returned in the table produced by SQLGetTypeInfo will be the name most commonly used by the data source.

SQL_ALL_EXCEPT_LIKE will be returned in the SEARCHABLE column for the Byte, Counter, Double, Single, Long, and Short data types. (The LIKE capability can be achieved by converting the value to a character using the ODBC canonical conversion functions, then performing the comparison.)

When the Text driver is used, SQLGetTypeInfo returns a CASE_SENSITIVE value of FALSE for the text data types (CHAR and LONGCHAR), when the data...
types actually are case-sensitive.

© 2016 Microsoft

SQLSetConnectOption (Text File Driver)

Note

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Option</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESS_MODE</td>
<td>The SQL_ACCESS_MODE option can be set to either SQL_MODE_READ_ONLY or SQL_MODE_READ_WRITE. However, the driver does not prevent updates if SQL_ACCESS_MODE is set to SQL_MODE_READ_ONLY.</td>
</tr>
<tr>
<td>SQL_AUTOCOMMIT</td>
<td>The Text driver only supports SQL_AUTOCOMMIT being set to ON (the default state), because they do not support transactions.</td>
</tr>
<tr>
<td>SQL_CURRENT_QUALIFIER</td>
<td>Supported.</td>
</tr>
<tr>
<td>SQL_LOGIN_TIMEOUT</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_OPT_TRACE</td>
<td>Supported.</td>
</tr>
<tr>
<td>SQL_OPT_TRACEFILE</td>
<td>Supported.</td>
</tr>
<tr>
<td>SQL_PACKET_SIZE</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_QIET_MODE</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_TRANSLATE_DLL</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_TRANSLATION_OPTION</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_TXN_ISOLATION</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

SQLStatistics (Text File Driver)

Note

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_QUALIFIER</td>
<td>The path to a directory. Pattern matching is not supported in the szTableQualifier argument.</td>
</tr>
<tr>
<td>TABLE_OWNER</td>
<td>NULL is returned in this column because owner name is not supported.</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>Undelimited table name. Pattern matching is not supported in the szTableName argument.</td>
</tr>
<tr>
<td>INDEX_QUALIFIER</td>
<td>NULL is always returned.</td>
</tr>
<tr>
<td>INDEX_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Only SQL_TABLE_STAT or SQL_INDEX_OTHER will be returned for TYPE.</td>
</tr>
<tr>
<td>SEQ_IN_INDEX</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>Index-dependent.</td>
</tr>
<tr>
<td>COLLATION</td>
<td>Index-dependent.</td>
</tr>
</tbody>
</table>
Filtering is based on uniqueness (the fUnique argument). The fAccuracy parameter is ignored.

## SQLTables (Text File Driver)

### Note

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>szTableOwner</td>
<td>The only valid argument for szTableOwner is NULL because none of the drivers support owner names. With szTableOwner set to NULL, all tables are returned. NULL is returned in the TABLE_OWNER column.</td>
</tr>
<tr>
<td>szTableQualifier</td>
<td>In the TABLE_QUALIFIER column, SQLTables will return the path to a directory.</td>
</tr>
<tr>
<td>SzTableType</td>
<td>&quot;TABLE&quot; is the only table type supported. When the Text driver is used, the list of files returned by SQLTables is determined by the file extensions in the Extensions List box in the ODBC Text Setup dialog box.</td>
</tr>
</tbody>
</table>

## SQLTransact (Text File Driver)

### Note

This topic provides Text File Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

This function supports SQL_COMMIT, but not SQL_ROLLBACK.

## Creating and Opening Tables (Text File Driver)

When the Text driver is used, a new table is created using the format specified in Odbcinst.ini. If not specified, tables are created in CSVDELIMITED format. By default, INTEGER columns default to 11 characters and FLOAT columns default to 22 characters. DATE columns use the YYYY-MM-DD format. CHAR and LONGCHAR columns are the width specified in the CREATE statement.

## Read–Only Status (Text File Driver)

When the Text driver is used, data source tables are read-only. However, applications can perform insertions for Text tables.
Additional Supported ODBC SQL Grammar

The ODBC drivers completely support the Minimum ODBC 2.5 grammar and the following clauses in the Core and Extended ODBC 2.5 grammar. Additional information can be displayed by jumping to the ALTER TABLE Statement, Correlation Names, and Procedure Invocation topics.

- Approximate numeric literal (Core)
- ALTER TABLE Statement (Core)
- Binary literals (Extended)
- Correlation Names (Core)
- Procedure Invocation (Extended)
- BETWEEN Predicate (Core)
- CREATE INDEX Statement
- Date Arithmetic
- Jet: Date, Time, and Timestamp Literals
- DROP INDEX Statement
- Fixed-Width Text File
- GROUP BY expression-list
- ORDER BY expression-list
- ORDER BY with GROUP BY
- EXISTS predicate (Core)
- IN (valuelist) (Core)
- Jet: Outer Joins (Extended)
- SELECT statement (Extended) (including GROUP BY, HAVING, and UNION clauses)
- Subqueries (SUBSELECTs) (Core)
- Scalar Functions
- Table Names

ALERT TABLE Statement

When the Microsoft Access, dBASE, or Paradox driver is used, the ALTER TABLE statement is supported with add and drop table constraint definition clauses. (ALTER TABLE statements are not supported for the Microsoft Excel or Text drivers.)

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you use the Paradox driver without implementing the Borland Database Engine, ALTER TABLE statements are not supported; only read and append statements are allowed.</td>
</tr>
</tbody>
</table>

Correlation Names
Correlation names are fully supported, including within the table list. For example, in the following string, E1 is the correlation name for the table named Emp:

```sql
SELECT * FROM Emp E1
WHERE E1.LastName = 'Smith'
```

---

**Procedure Invocation**

When the Microsoft Access driver is used, procedures can be invoked from the driver by using the `SQLExecDirect` or `SQLPrepare` function with the following syntax: `{CALL procedure-name [(parameter,parameter) ...]}`. Note that expressions are not supported as parameters to a called procedure.

If a procedure name includes a dash, the name must be delimited with back quotes (``).

A parameterized query can be called using the previous statement.

---

**BETWEEN Predicate**

The syntax:

```
expression1 BETWEEN expression2 AND expression3
```

returns true only if `expression1` is greater than or equal to `expression2` and `expression1` is less than or equal to `expression3`.

The semantics of this syntax are different for the Desktop Database Drivers and the Microsoft Jet engine. In Microsoft Jet SQL, `expression2` can be greater than `expression3` so that the statement will return TRUE only if `expression1` is greater than or equal to `expression3`, and `expression1` is less than or equal to `expression2`.

---

**CREATE INDEX Statement**

The syntax of the CREATE INDEX statement is:

```
CREATE [UNIQUE] INDEX index-name ON table-name (column-identifier [ASC] [DESC], column-identifier [ASC] [DESC] ... ) WITH <index option list>
```

where `<index option list>` can be: PRIMARY | DISALLOW NULL | IGNORE NULL

Only the Microsoft Access driver uses the DISALLOW NULL and IGNORE NULL index options. The dBASE and Paradox drivers accept the syntax, but ignore the presence of either option.

When the Paradox driver is used, the CREATE INDEX statement creates Paradox primary key files and secondary files.

This statement is not supported by the Microsoft Excel or Text drivers.

---

**Date Arithmetic**

The driver supports adding and subtracting an integer from a DATE, TIME, or TIMESTAMP column. For a DATE column, the integer specifies the number of days to add or subtract. For a TIME or TIMESTAMP column, the integer specifies the number of seconds to add or subtract.
Jet: Date, Time, and Timestamp Literals

For maximum interoperability, applications should pass date literals in the ODBC canonical format using escape-clause syntax:

- For date literals, `{d 'value'}`, where value is in the form "yyyy-mm-dd"
- For time literals, `{t 'value'}`, where value is in the form "hh:mm:ss"
- For timestamp literals `{ts 'value'}`, where value is in the form "yyyy-mm-dd hh:mm:ss[.f...]".

DROP INDEX Statement

When the Microsoft Access, dBASE, or Paradox driver is used, the syntax of the DROP INDEX statement is "DROP INDEX a on b" where "a" is the name of the index and "b" is the name of the table (not DROP INDEX index-name).

When the Paradox driver is used, the DROP INDEX statement deletes Paradox secondary index files.

The DROP INDEX statement is not supported for the Microsoft Excel or Text drivers.

Fixed-Width Text File

When the Text driver is used, the last column of a fixed-width text file can be variable length.

GROUP BY expression-list

GROUP BY supports an expression list as well as a column name. The select list of a SELECT statement that has a GROUP BY clause can only include expressions from the GROUP BY clause or set functions.
ORDER BY expression-list

Expressions can be used in the ORDER BY clause. For example, in the following clauses the table is ordered by three key expressions: a+b, c+d, and e.

```
SELECT * FROM emp
ORDER BY a+b, c+d, e
```

No ordering is allowed on set functions or an expression that contains a set function.

ORDER BY with GROUP BY

ORDER BY can be performed on any expression in the GROUP BY expression-list or any column in the result set.

Jet: Outer Joins

A SELECT statement can contain a list of OUTER JOIN clauses. Nested OUTER JOINs are supported. For more information about OUTER JOINs, see the Microsoft Jet Database Engine Programmer’s Guide.

Scalar Functions

The Microsoft ODBC Desktop Database Drivers support the following scalar functions:

<table>
<thead>
<tr>
<th>CONCAT</th>
<th>LCASE</th>
<th>RIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONVERT</td>
<td>LEFT</td>
<td>RTRIM</td>
</tr>
<tr>
<td>CURDATE</td>
<td>LENGTH</td>
<td>SUBSTRING</td>
</tr>
<tr>
<td>CURTIME</td>
<td>LOCATE</td>
<td>UCASE</td>
</tr>
</tbody>
</table>
For information about the arguments and return values of scalar functions, see Appendix F of the ODBC Programmer’s Reference.

## Table Names

When the dBASE, Microsoft Excel, Paradox, or Text driver is used, table names that occur in the FROM clause of SELECT or DELETE, after the INTO clause in INSERT, and after UPDATE, CREATE TABLE, and DROP TABLE can contain a valid path, primary name, and file name extension.

Use of a table name elsewhere in an SQL statement does not support the use of paths or extensions but will accept only the primary name (for example, EMP FROM C:\ABC\EMP).

Correlation names (aliases) can be used. For example:

```sql
SELECT * FROM C:\ABC\EMP T1 WHERE T1.COL1 = 'aaa'
```

## Limitations

This section describes limitations of the ODBC Desktop Database Drivers with regard to the following topics:

<table>
<thead>
<tr>
<th>Functions</th>
<th>Statements</th>
<th>Clauses, types, and so on</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate Function</td>
<td>ALTER TABLE Statement</td>
<td>Column Name</td>
</tr>
<tr>
<td>Scalar Function</td>
<td>CALL Statement</td>
<td>Data Type</td>
</tr>
<tr>
<td>Set Functions</td>
<td>CREATE INDEX Statement</td>
<td>Date Arithmetic</td>
</tr>
<tr>
<td>Sorting</td>
<td>CREATE TABLE Statement</td>
<td>Identifiers</td>
</tr>
<tr>
<td>CONVERT Function</td>
<td>DELETE Statement</td>
<td>Index Name</td>
</tr>
<tr>
<td></td>
<td>DROP INDEX Statement</td>
<td>Parameterized Query</td>
</tr>
<tr>
<td></td>
<td>DROP TABLE Statement</td>
<td>Reserved Words</td>
</tr>
<tr>
<td></td>
<td>INSERT Statement</td>
<td>AND Predicate</td>
</tr>
<tr>
<td></td>
<td>SELECT DISTINCT Statement</td>
<td>DISTINCT Keyword</td>
</tr>
<tr>
<td></td>
<td>SELECT Statement</td>
<td>FROM Clause</td>
</tr>
<tr>
<td></td>
<td>UPDATE Statement</td>
<td>HAVING Clause</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LIKE Predicate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOT NULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ORDER BY Clause</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table Name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table References</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Views</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WHERE Clause</td>
</tr>
</tbody>
</table>
Aggregate Function Limitations

An aggregate function and a non-aggregate column reference cannot both be used as arguments to a single SQL statement.

Column Name Limitations

Column names can contain any valid characters (for example, spaces). If column names contain any characters except letters, numbers, and underscores, the name must be delimited by enclosing it in back quotes (`)

When the Microsoft Access or Microsoft Excel driver is used, column names are limited to 64 characters, and longer names generate an error. When the Paradox driver is used, the maximum column name is 25 characters. When the Text driver is used, the maximum column name is 64 characters, and longer names are truncated.

When the dBASE driver is used, characters with an ASCII value greater than 127 are converted to underscores.

When the Microsoft Excel driver is used, if column names are present, they must be in the first row. A name that in Microsoft Excel would use the "!" character must be enclosed in back quotes (``). The "!" character is converted to the "$" character, because the "!" character is not legal in an ODBC name, even when the name is enclosed in back quotes. All other valid Microsoft Excel characters (except the pipe character (|)) can be used in a column name, including spaces. A delimited identifier must be used for a Microsoft Excel column name to include a space. Unspecified column names will be replaced with driver-generated names, for example, "Col1" for the first column.

The pipe character (|) cannot be used in a column name, whether the name is enclosed in back quotes or not.

When the Text driver is used, the provider provides a default name if a column name is not specified. For example, the driver calls the first column F1, the second column F2, and so on.

Data Type Limitations

The Microsoft ODBC Desktop Database Drivers impose the following limitations on data types:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All data types</td>
<td>Type conversion failures might result in the affected column being set to NULL.</td>
</tr>
<tr>
<td>BINARY</td>
<td>Creating a zero-length BINARY column actually returns a 255-byte BINARY column.</td>
</tr>
<tr>
<td>DATE</td>
<td>The DATE data type cannot be converted to another data type (or itself) by the CONVERT function.</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>Supports maximum precision and a scale of 28.</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>Supports maximum precision and a scale of 28.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The TIMESTAMP data type cannot be converted to itself by the CONVERT function.</td>
</tr>
</tbody>
</table>
TINYINT values are always unsigned.

Zero-Length Strings
When a dBASE, Microsoft Excel, Paradox, or Textdriver is used, inserting a zero-length string into a column actually inserts a NULL instead.

Date Arithmetic Limitations

Date arithmetic is not supported for subtracting a DATE data type from a DATE data type.

Identifiers Limitations

If an identifier contains a space or a special symbol, the identifier must be enclosed in back quotes. A valid name is a string of no more than 64 characters, of which the first character must not be a space. Valid names cannot include control characters or the following special characters: ` | # * ? [ ] . ! $ .

Do not use the reserved words listed in the SQL grammar in Appendix C of the ODBC Programmer's Reference (or the shorthand form of these reserved words) as identifiers (that is, table or column names), unless you surround the word in back quotes (`).

Index Name Limitations

When the Paradox driver is used, a primary index must have the same name as the table upon which it is defined. Other unique or non-unique indexes must have the same name as the table upon which they are defined.

Parameterized Query Limitations

When the Microsoft Access driver is used, a parameterized query can be called using the following syntax: CALL query-name [(parameter[,parameter]...)].
Reserved Word Limitations

Do not use the reserved words listed in the SQL grammar in Appendix C of the ODBC Programmer's Reference (or the shorthand form of these reserved words) as identifiers (that is, table or column names), unless you surround the word in back quotes (``).

Scalar Function Limitations

Scalar functions are supported only by using the ODBC canonical format.

Set Functions Limitations

The set functions (AVG, MAX, MIN, and SUM) do not support the DISTINCT keyword.

Sorting Limitations

The maximum length of a sort key in a GROUP BY clause, ORDER BY clause, SELECT DISTINCT statement, or outer join is 255 bytes; the maximum length of all sort keys in a sort row is 65,500 bytes.
String Limitations

The maximum length of an SQL statement string is 65,000 characters.

When the Microsoft Access driver is used, only SQL-92 string constants (with single quotation marks, not double quotation marks) are supported.

The pipe character (|) cannot be used in a string, whether the character is enclosed in back quotes or not.

For maximum interoperability, applications should pass strings in parameters, rather than passing quoted strings.

Table Name Limitations

Table names can contain any valid characters (for example, spaces). If table names contain any characters except letters, numbers, and underscores, the name must be delimited by enclosing it in back quotes (').

When the Microsoft Excel driver is used, and a table name is not qualified by a database reference, the default database is implied. If a name in Microsoft Excel includes the "!" character, it will automatically be translated to the "$" character instead.

The Microsoft Excel table name that references <filename> is supported for Microsoft Excel 3.0 and 4.0 files. The Microsoft Excel table name that references <workbook-name> is supported for Microsoft Excel 5.0, 7.0, or 97 files.

When the dBASE driver is used, characters with an ASCII value greater than 127 are converted to underscores.

When the Microsoft Access driver is used, the table name is limited to 64 characters.

When the dBASE, Microsoft Excel 3.0 or 4.0, Paradox, or Text driver is used, special MS-DOS keywords CON, AUX, LPT1, and LPT2 should not be used as table names.

Table References Limitations

A maximum of 16 table references can be included in any query statement.

Views Limitations

Not supported by the dBASE, Microsoft Excel, Paradox, or Text drivers.
ALTER TABLE Statement Limitations

When the dBASE or Paradox driver is used, once an index has been created and a new record added, the structure of the table cannot be changed by the ALTER TABLE statement unless the index is dropped and the contents of the table are deleted.

ALTER TABLE statements are not supported for the Microsoft Excel or Text drivers.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you use the Paradox driver without implementing the Borland Database Engine, ALTER TABLE statements are not supported; only read and append statements are allowed.</td>
</tr>
</tbody>
</table>

AND Predicate Limitations

A maximum of 40 is supported.

CALL Statement Limitations

(This limitation applies to the Microsoft Access driver only.)

Expressions are not supported as parameters to a called procedure.

CONVERT Function Limitations

Type conversion failures result in the affected column being set to NULL.

Neither the DATE nor TIMESTAMP data type can be converted to another data type (or itself) by the CONVERT function.

CREATE INDEX Statement Limitations
The CREATE INDEX statement is not supported for the Microsoft Excel or Text drivers.

An index can be defined on a maximum of 10 columns. If more than 10 columns are included in a CREATE INDEX statement, the index will not be recognized and the table will be treated as though no index were created.

The dBASE driver cannot create an index on a LOGICAL column.

When the dBASE driver is used, response time on large files can be improved by building an .mdx (or .ndx) index on the column (field) specified in the WHERE clauses of a SELECT statement. Existing .mdx indexes will automatically be applied for =, >, <, >=, =<, and BETWEEN operators in a WHERE clause, and LIKE predicates, as well as in join predicates.

When the dBASE driver is used, the index created by a CREATE UNIQUE INDEX statement is actually non-unique, and duplicate values can be inserted into the indexed column. Only one record from a set with identical key values can be added to the index.

When the Paradox driver is used, a unique index must be defined upon a contiguous subset of the columns in a table, including the first column. A table cannot be updated by the Paradox driver if a unique index is not defined on the table or when the Paradox driver is used without the implementation of the Borland Database Engine.

CREATE TABLE Statement Limitations

When the Microsoft Access, Microsoft Excel, or Paradox driver is used, and the length of a text or binary column is not specified (or is specified as 0), the column length will be set to 255.

When the dBASE driver is used, and the length of a text or binary column is not specified (or is specified as 0), the column length will be set to 254.

A maximum of 255 columns is supported.

When the Microsoft Excel driver is used on a MicrosoftExcel 5.0, 7.0, or 97 data source, a worksheet cannot be created with the same name as a worksheet that was previously dropped. When the Microsoft Excel driver is used to access a version 5.0, 7.0, or 97 worksheet, a DROP TABLE statement clears the worksheet, but does not delete the worksheet name.

When the Paradox driver is used, columns cannot be added once an index has been defined on a table. If the first column of the argument list of a CREATE TABLE statement creates an index, a second column cannot be included in the argument list.

DELETE Statement Limitations

The DELETE statement is not supported for the Microsoft Excel or Text driver. Note that the INSERT statement is supported for the Text driver.

The dBASE driver does not support packing a table to remove "deleted" values.

For the Paradox driver to delete a row from a table, the table must have a unique index (Paradox primary key).

DISTINCT Keyword Limitations

Not supported for Long Text fields (Microsoft Access) or Memo fields (dBASE).

DROP INDEX Statement Limitations
The DROP INDEX statement is not supported for the Microsoft Excel or Text driver.

**DROP TABLE Statement Limitations**

When the Microsoft Excel 5.0, 7.0, or 97 driver is used, the DROP TABLE statement clears the worksheet but does not delete the worksheet name. Because the worksheet name still exists in the workbook, another worksheet cannot be created with the same name.

**FROM Clause Limitations**

The maximum number of tables in a FROM clause is 16.

**HAVING Clause Limitations**

The maximum number of search conditions in a HAVING clause is 40.

**INSERT Statement Limitations**

Inserted data is truncated on the right without warning if it is too long to fit into the column.
Attempting to insert a value that is out of the range of a column’s data type causes a NULL to be inserted into the column.

When a dBASE, Microsoft Excel, Paradox, or Textdriver is used, inserting a zero-length string into a column actually inserts a NULL instead.

When the Microsoft Excel driver is used, if an empty string is inserted into a column, the empty string is converted to a NULL; a searched SELECT statement that is executed with an empty string in the WHERE clause will not succeed on that column.

A table is not updatable by the Paradox driver under two conditions:

- When a unique index is not defined on the table. This is not true for an empty table, which can be updated with a single row even if a unique index is not defined on the table. If a single row is inserted in an empty table that does not have a unique index, an application cannot create a unique index or insert additional data after the single row has been inserted.

- If the Borland Database Engine is not implemented, only read and append statements are allowed on the Paradox table.

When the Text driver is used, NULL values are represented by a blank-padded string in fixed-length files, but are represented by no spaces in delimited files. For example, in the following row containing three fields, the second field is a NULL value:

```
"Smith:,, 123
```

When the Text driver is used, all column values can be padded with leading spaces. The length of any row must be less than or equal to 65,543 bytes.

© 2016 Microsoft

**LIKE Predicate Limitations**

If data in a column is longer than 255 characters, the LIKE comparison will be based only on the first 255 characters.

A LIKE used in a procedure is supported only with constant patterns. The Desktop Database Drivers support SQL-92 LIKE pattern matching.

Use of an escape clause in a LIKE predicate is not supported.

A LIKE comparison should not be performed on a column containing data of a numeric or float data type. The results may be unpredictable. For more information, see the Microsoft Jet Database Engine Programmer’s Guide.

© 2016 Microsoft

**NOT NULL Limitations**

The NOT NULL constraint in the CREATE TABLE statement is not supported.

© 2016 Microsoft

**ORDER BY Clause Limitations**

If a SELECT statement contains a GROUP BY clause and an ORDER BY clause, the ORDER BY clause can contain only a column in the result set or an expression in the GROUP BY clause.
SELECT DISTINCT Limitations

The DISTINCT keyword does not apply to binary data.

SELECT Statement Limitations

An aggregate-function column cannot be mixed with a non-aggregate column in a SELECT statement.

The select list of a SELECT statement that has a GROUP BY clause can only have expressions from the GROUP BY clause or set functions.

The use of an asterisk (to select all columns) in a SELECT statement containing a GROUP BY clause is not supported. The names of the columns to be selected must be specified.

The use of a vertical bar in a SELECT statement is not supported. Use a parameter in the SELECT statement if you need to refer to a data value that contains a vertical bar.

When using a column alias in a SELECT statement, the word "as" must precede the alias. For example, "SELECT col1 as a from b." Without the "as", the statement will return an error.

If an incorrect column name is entered into a SELECT statement, a SQLSTATE 07001 error, "Wrong Number of Parameters," is returned instead of a SQLSTATE 50022 error, "Column Not Found."

When the Microsoft Excel driver is used, if an empty string is inserted into a column, the empty string is converted to a NULL; a searched SELECT statement that is executed with an empty string in the WHERE clause will not succeed on that column.

UPDATE Statement Limitations

For the Paradox driver to update a table, the table must have a unique index (Paradox primary key). When you use the Paradox driver without implementing the Borland Database Engine, it is not possible to update a Paradox table.

Not supported by the Text driver.

When the Microsoft Excel driver is used, it is possible to update values, but a row cannot be deleted from a table based on a Microsoft Excel spreadsheet. As a result, the UPDATE statement is not considered officially supported by the Microsoft Excel driver. Only the INSERT statement is considered supported.

WHERE Clause Limitations

The maximum number of clauses in a WHERE clause is 40.

LONGVARBINARY and LONGVARCHAR columns can be compared to literals of up to 255 characters in length, but cannot be compared using parameters.
WHERE CURRENT OF Clause Limitations

Not supported.

ODBC Errors

When an error occurs, the Microsoft ODBC Desktop Database Drivers return the native error number, the SQLSTATE (an ODBC error code), and an error message.

This section contains the following topics.

- Native Error
- SQLSTATE
- Error Messages
- Arithmetic Errors

Native Error

Positive native error codes are generated by the Desktop Database Drivers. Negative error codes are generated by Microsoft Jet. For more information, see the Microsoft Jet Database Engine Programmer’s Guide.

SQLSTATE

For errors that occur in the data source, the ODBC driver maps the returned native error to the appropriate SQLSTATE. For errors that are detected by the driver or the Driver Manager, the ODBC driver or Driver Manager generates the appropriate SQLSTATE.
ODBC Jet Error Messages

For errors that occur in the data source, the ODBC driver returns an error message returned to it by the ODBC File Library. For errors that occur in the ODBC driver or the Driver Manager, the driver returns an error message based on the text associated with the SQLSTATE.

Error messages have the following format:

\[
[vendor][ODBC-component][data-source]message-text
\]

The prefixes in brackets ([ ]) identify the location of the error. When the error occurs in the Driver Manager, data-source is not given. When the error occurs in the data source, the [vendor] and [ODBC-component] prefixes identify the vendor and name of the ODBC component that received the error from the data source.

The following table shows the error messages returned by the Driver Manager and driver ISAM:

<table>
<thead>
<tr>
<th>Error message</th>
<th>Error location</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Microsoft][ODBC Driver Manager] message-text</td>
<td>Driver Manager (Odbc32.dll)</td>
</tr>
<tr>
<td>[Microsoft][ODBC driver-name]message-text</td>
<td>Driver ISAM (see Driver ISAMs)</td>
</tr>
</tbody>
</table>

Arithmetic Errors

The ODBC driver evaluates the WHERE clause in a SELECT statement as it fetches each row. If a row contains a value that causes an arithmetic error, such as divide-by-zero or numeric overflow, the driver returns all rows, but returns errors for columns with arithmetic errors. When inserting or updating, however, the ODBC driver stops inserting or updating data when the first arithmetic error is encountered.

Supported ODBC API Functions

The purpose of leveling is to inform the application what features are available to it from the driver. The Microsoft ODBC Desktop Database Drivers support all Core and Level 1 functions.

For more information about conformance levels for functions and grammar, see Conformance Levels in the ODBC Programmer's Reference.

Support of ODBC API functions can be dependent on the driver used. The following table summarizes the support for functions. The leftmost column provides a link to the general reference page for each function. These reference pages are listed alphabetically in the ODBC API Reference section, under ODBC Programmer's Reference. The columns to the right provide links to driver-specific notes about each supported function. These driver-specific topics are listed in the "Other Programming Details" section for each driver. Alternatively, if the same remarks about a function apply to all the ODBC Desktop Database Drivers, the rightmost column provides a link to a topic that summarizes the Desktop Database Drivers' support for that function. These topics are listed at the end of the current section ("Supported ODBC API Functions").

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLBindParameter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Excel</td>
</tr>
<tr>
<td>SQLColAttributes</td>
<td>Access</td>
<td>dBASE</td>
<td>Paradox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLColumns</td>
<td>Access</td>
<td>dBASE</td>
<td>Paradox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Access</td>
<td>dBASE</td>
<td>Paradox</td>
<td>Text</td>
<td>Excel</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>--------</td>
<td>-------</td>
<td>---------</td>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>SQLConfigDataSource</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLDriverConnect</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLGetCursorName</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLGetData</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLGetInfo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLGetStmtOption</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLGetTypeInfo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLGetTypeInfo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLMoreResults</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLPrepare</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLProcedureColumns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLProcedures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLSetConnectOption</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLSetCursorName</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLSetPos</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLSetScrollOptions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLSetStmtOption</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLSpecialColumns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLStatistics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLTables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLTransact</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following topics provide remarks about ODBC functions. These remarks apply to all ODBC Desktop Database Drivers.

- SQLGetData (Desktop Database Drivers)
- SQLGetStmtOption (Desktop Database Drivers)
- SQLMoreResults (Desktop Database Drivers)
- SQLPrepare (Desktop Database Drivers)
- SQLProcedures (Desktop Database Drivers)
- SQLSetCursorName (Desktop Database Drivers)
- SQLSetPos (Desktop Database Drivers)
- SQLSetScrollOptions (Desktop Database Drivers)
SQLGetCursorName (Desktop Database Drivers)

`SQLGetCursorName` is supported, but can only be used when the Cursor Library is used, because positioned operations are not supported in the driver.

**Note**

*ppcbValue in SQLGetData may return twice as many characters as actually available when binding to ANSI data longer than 510 characters on a Jet 4.0 database. Character values of 510 or less will return the actual cbValue.*

SQLGetData (Desktop Database Drivers)

This function can retrieve data from any column, whether or not there are bound columns after it and regardless of the order in which the columns are retrieved.

SQLGetStmtOption (Desktop Database Drivers)

The bookmarks returned by an fOption of SQL_GETBOOKMARK are only valid while the query is open and are invalidated when the query is reissued. Persistent bookmarks are not supported.

SQLMoreResults (Desktop Database Drivers)

This function always returns SQL_NO_DATA_FOUND.
SQLPrepare (Desktop Database Drivers)

A statement prepared with an incorrect number of columns returns an error at execution time, not upon statement preparation.

SQLProcedures (Desktop Database Drivers)

_SQLProcedures_ will only return rows for those procedures that have at least one argument. Procedures that have no arguments are treated as views.

<table>
<thead>
<tr>
<th>Column</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE_QUALIFIER</td>
<td>The path to the database file.</td>
</tr>
<tr>
<td>PROCEDURE_OWNER</td>
<td>NULL</td>
</tr>
<tr>
<td>PROCEDURE_NAME</td>
<td>Undelimited procedure name</td>
</tr>
<tr>
<td>PROCEDURE_TYPE</td>
<td>SQL_PT_PROCEDURE</td>
</tr>
</tbody>
</table>

SQLSetCursorPosition (Desktop Database Drivers)

Because the driver does not support a positioned update or delete by the WHERE CURRENT OF _columnName_ syntax, _SQLSetCursorPosition_ is supported, but cannot be used for positioned updates. It can only be used when the Cursor Library is enabled and the application is using _SQLExtendedFetch_.

SQLSetPos (Desktop Database Drivers)

The bulk-model semantics for _SQLSetPos_ calls with the _irow_ argument equal to 0 are supported. _SQL_LOCK_NO_CHANGE_ is supported for _fLock_. _SQL_LOCK_EXCLUSIVE_ and _SQL_LOCK_UNLOCK_ are not supported. _SQLSetPos_ supports updatable joins. (For more information, see the Microsoft Jet Database Engine Programmer's Guide.)
SQLSetScrollOptions (Desktop Database Drivers)

Forward and static cursors are supported for SQL_CONCUR_READ_ONLY.

Only keyset-driven cursors are supported for an fConcurrency argument of SQL_CONCUR_LOCK.

An fConcurrency argument of SQL_CONCUR_ROWVER is not supported.

Dynamic cursors and mixed cursors are not supported.

SQLSetStmtOption (Desktop Database Drivers)

<table>
<thead>
<tr>
<th>fOption</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ASYNC_ENABLE</td>
<td>Asynchronous processing is not supported. The SQL_ASYNC_ENABLE fOption will return SQLSTATE S1C00 (Driver not capable).</td>
</tr>
<tr>
<td>SQL_KEYSET_SIZE</td>
<td>The only valid keyset size is 0, because mixed and dynamic cursors are not supported. If this value is set to any other number, it will be changed to 0 and the call will return SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed).</td>
</tr>
<tr>
<td>SQL_MAX_ROWS</td>
<td>The only valid rowset size is 0, because the Desktop Database Drivers do not support limiting the number of rows that are returned. If this value is set to any other number, it will be changed to 0 and the call will return SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed).</td>
</tr>
<tr>
<td>SQL_QUERY_TIMEOUT</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_ROW_NUMBER</td>
<td>Not supported.</td>
</tr>
<tr>
<td>SQL_SIMULATE_CURSOR</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>

SQLSpecialColumns (Desktop Database Drivers)

A unique index will be returned (if one exists) for the SQL_BEST_ROWID flag in fColType. No result set will be returned for the SQL_ROWVER flag.

All row IDs have a scope of SQL_SCOPE_CURROW.

Pattern matching is not supported for either the szTableQualifier or szTableName argument.
Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The Microsoft® ODBC Driver for Oracle allows you to connect your ODBC-compliant application to an Oracle database. The ODBC Driver for Oracle conforms to the Open Database Connectivity (ODBC) specification described in the ODBC Programmer’s Reference. It allows access to PL/SQL packages, XA/DTC integration, and Oracle access from within Internet Information Services (IIS).

Oracle RDBMS is a multiuser relational database management system that runs with various workstation and minicomputer operating systems. IBM-compatible computers running Microsoft Windows can communicate with Oracle database servers over a network. Supported networks include Microsoft LAN Manager, NetWare, VINES, DECnet, and any network that supports TCP/IP.

The ODBC Driver for Oracle enables an application to access data in an Oracle database through the ODBC interface. The driver can access local Oracle databases or it can communicate with the network through SQL*Net. The following diagram details this application and driver architecture.

The ODBC Driver for Oracle complies with API Conformance Level 1 and SQL Conformance Level Core. It also supports some functions in API Conformance Level 2 and most of the grammar in the Core and Extended SQL conformance levels. The driver is ODBC 2.5 compliant and supports 32-bit systems. Oracle 7.3x is supported fully; Oracle8 has limited support. The ODBC Driver for Oracle does not support any of the new Oracle8 data types — Unicode data types, BLOBs, CLOBs, and so on — nor does it support Oracle's new Relational Object Model. For more information about supported data types, see Supported Data Types in this guide.

To access Oracle data, the following components are required:

- The ODBC Driver for Oracle
- An Oracle RDBMS database
- Oracle Client Software

Additionally, for remote connections:

- A network that connects the computers that run the driver and the database. The network must support SQL*Net connections.

Component Documentation

This guide contains detailed information about setting up and configuring the Microsoft ODBC Driver for Oracle and adding programmatic functionality. It also contains technical reference material.

For information regarding specific Oracle product behavior, consult the documentation that accompanies the Oracle product.

For information about setting up or configuring the Microsoft ODBC Driver for Oracle using the ODBC Data Source Administrator, see the ODBC Data Source Administrator documentation.
This section contains the following topics.

- ODBC Driver for Oracle User's Guide
- ODBC Driver for Oracle Programmer's Reference

© 2016 Microsoft

ODBC Driver for Oracle User's Guide

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The ODBC Driver for Oracle User's Guide is intended to complement the setup help that can be accessed through the ODBC Data Source Administrator.

This section contains the following topics.

- Driver Version Scheme
- Installing the Software
- Oracle Software Patches
- Adding and Modifying Data Sources Using Setup
- Testing the ODBC Connection
- Configuring the ODBC Driver for Oracle
- Connecting to a Data Source
- Determining Installed Oracle Components
- Setting the Date Format on Connection

© 2016 Microsoft

Driver Version Scheme

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The following table lists all released versions of the Microsoft ODBC Driver for Oracle.

<table>
<thead>
<tr>
<th>Driver version</th>
<th>Build number</th>
<th>Availability history</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.00.6235</td>
<td>Visual C++ 4.2 and Visual Basic 5.0, Enterprise Edition</td>
</tr>
<tr>
<td>2.0</td>
<td>2.73.7269</td>
<td>Visual Studio 97 and MDAC 1.5a</td>
</tr>
<tr>
<td>2.0 updated</td>
<td>2.73.7283.01</td>
<td>IIS 4.0</td>
</tr>
<tr>
<td>2.0 updated</td>
<td>2.73.7283.03</td>
<td>MDAC 1.5b and 1.5c</td>
</tr>
<tr>
<td>2.0 updated</td>
<td>2.73.7356</td>
<td>ODBC 3.5 SDK</td>
</tr>
<tr>
<td>2.5</td>
<td>2.573.2927</td>
<td>Visual Studio 6.0 and MDAC 2.0</td>
</tr>
<tr>
<td>2.5 updated</td>
<td>2.573.3513</td>
<td>SQL Server 7.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL Server 6.5 SPS</td>
</tr>
</tbody>
</table>

Build 2.00.6235 (version 1) was the first release of the Microsoft ODBC Driver for Oracle. After the release of the first version, a new naming convention was adopted.
For example, 2.73.7283.03 can be divided into the following distinct components:

- 2 = The version number.
- 73 = The version of Oracle Server for which the driver was designed.
- 7283.03 = The build number of the driver.

Note

With release 2.573.2973, the naming convention has led to some confusion that 2.573 is an earlier release than 2.73, but each section of the build number should be considered individually. The number 573 is larger than 73, so it is a newer version. Also, “2.5” indicates the driver’s version number.

© 2016 Microsoft

Installing the Software (ODBC)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The ODBC Driver for Oracle is one of the data access components. It accompanies other ODBC components, such as the ODBC Data Source Administrator, and should already be installed. The driver also can be found under "Drivers and Other Downloads" on the Microsoft Product Support Services Online Web site at www.microsoft.com.

Network software must be installed according to its own documentation. The ODBC Driver for Oracle requires no special installation considerations as long as the network software is supported.

Oracle software must be installed according to its own documentation. The ODBC Driver for Oracle generally requires no special installation considerations as long as the driver supports the version. However, to keep products compatible, install the ODBC Driver for Oracle last to ensure you have the latest version of the driver. Oracle maintains a public FTP site where it posts, among other things, patches to the Oracle server products and the client component that ships with the server products. These patches are required for the proper functioning of several Microsoft products and technologies. For more information about this site, see Oracle Software Patches.

Caution

Installing Oracle software over MDAC/Windows DAC may overwrite current versions of MDAC. If problems arise using ODBC components, reinstall MDAC.

© 2016 Microsoft

Oracle Software Patches

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

Patches for the Oracle server products and its client component are required for the proper functioning of several Microsoft products and technologies, including the Microsoft ODBC Driver for Oracle, the Microsoft OLE DB Provider for Oracle, Internet Information Services (IIS), Component Services (or Microsoft Transaction Server, if you are using Windows NT), and so forth.

Note

The following instructions may not be completely accurate because the Oracle FTP site is subject to change.

To download the Oracle software patches

1. Connect to the public FTP site at oracle-ftp.oracle.com. The user ID is "anonymous" and the password is your e-mail address.
2. Navigate to the following directory: /server/wgt_tech/server/windowsNT.
3. To download patches most relevant for Windows 95, Windows 98 and Windows NT/Windows 2000, navigate to the subdirectory for your version of Oracle — 7.3 or 8.0. The two subdirectories are /73patchsets and /80patchsets.
4. To download patches for Oracle’s network technology, either SQL*Net or Net8, navigate to the following directory: /network.

Accessing this FTP site from your Web browser might not work. If you experience problems, try using a "traditional" FTP client or use the DOS command prompt.

Note
Because Oracle fixes bugs in current versions and then retrofits them to earlier versions using software patches, it is recommended that you download the latest patch available. This is especially true for the Oracle Server Client components. If you have questions about installing these patches, contact Oracle Support.

Adding and Modifying Data Sources Using Setup

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

A data source identifies a path to data that can include a network library, server, database, and other attributes — in this case, the data source is the path to an Oracle database. To connect to a data source, the Driver Manager checks the Windows registry for specific connection information.

The registry entry created by the ODBC Data Source Administrator is used by the ODBC Driver Manager and ODBC drivers. This entry contains information about each data source and its associated driver. Before you can connect to a data source, its connection information must be added to the registry.

To add and configure data sources, use the **ODBC Data Source Administrator**. The ODBC Administrator updates your data source connection information. As you add data sources, the ODBC Administrator updates the registry information for you.

**To add a data source for Windows**

1. Open the ODBC Data Source Administrator.
2. In the ODBC Data Source Administrator dialog box, click Add. The Create New Data Source dialog box appears.
3. Select Microsoft ODBC for Oracle and then click Finish. The Microsoft ODBC for Oracle Setup dialog box appears.
4. In the Data Source Name box, type the name of the data source you want to access. It can be any name that you choose.
5. In the Description box, type the description for the driver. This optional field describes the database driver that the data source connects to. It can be any name that you choose.
6. In the User Name box, type your database user name (your database user ID).
7. In the Server box, type the Database Alias or connect string for the Oracle Server engine that you want to access.
8. Click OK to add this data source.

**Note**

The Data Sources dialog box appears, and the ODBC Administrator updates the registry information. The user name and connect string that you typed become the default connection values for this data source when you connect to it.

1. Click Options make more specifications about the ODBC Driver for Oracle setup:
   - **Translation** — Click Select to choose a loaded data translator. The default is <No Translator>.
   - **Performance** — The Include REMARKS in Catalog Functions check box specifies whether the driver returns Remarks columns for the SQLColumns result set. The ODBC Driver for Oracle provides faster access when this value is not set. The Include SYNONYMS in SQL Columns check box specifies whether the driver returns column information. **Buffer Size** specifies the size, in bytes, allocated to receive fetched data. The driver optimizes fetching so that one fetch from the Oracle Server returns enough rows to fill a buffer of the specified size. Larger values tend to increase performance when fetching a lot of data.
   - **Customization** — The Enforce ODBC DayOfWeek Standard check box specifies whether the result set will conform to the ODBC specified day-of-week format (Sunday = 1; Saturday = 7). If this check box is cleared, the locale-specific Oracle value is returned. The SQLDescribeCol **always returns a value for precision** check box specifies whether or not the driver should return a non-zero value for the cbColDef argument of SQLDescribeCol. This connection string attribute applies only to columns where there is no Oracle-defined scale, such as computed numeric columns and columns defined as NUMBER without a precision or scale. A SQLDescribeCol call returns 130 for the precision when Oracle does not provide that information. If this check box is cleared, the driver will return 0 for these types of columns instead.
2. Click Add to add another data source, or click Close to exit.

**To modify a data source for Windows**

1. Open the ODBC Data Source Administrator. Click the appropriate DSN tab.
2. Select the Oracle data source you want to modify and then click Configure. The Microsoft ODBC for Oracle Setup dialog box appears.
3. Modify the applicable data source fields, and then click OK.

When you have finished modifying the information in this dialog box, the ODBC Administrator updates the registry information.
Testing the ODBC Connection

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

When troubleshooting ODBC access to Oracle 7.x and Oracle8 RDBMS servers, it might be necessary to verify that the underlying SQL*Net and Oracle Protocol Adapters are correctly installed. To do this, use the Oracle-supplied utility Nettest.exe in the Orawin\Bin directory.

Nettest is a simple utility that attempts to log on to the selected server using only the installed SQL*Net software that is part of the Oracle client. The utility will ask for a login name, password, and TNS connect string. If the Oracle client is correctly installed, the utility will simply display "Ping Successful." If the login was not successful, you will need to consult with a database administrator.

Configuring the ODBC Driver for Oracle

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

You can control performance of the ODBC Driver for Oracle by knowing the data environment and correctly setting the parameters of the data source connection through the ODBC Data Source Administrator dialog box or through connect string parameters. The dialog box provides the following controls for connecting to a data source using the dialog box or using connect strings:

- **User DSN tab** Lists the Data Source Names that are local to the computer.
- **System DSN tab** Enables you to add or remove a system data source. System data sources can be accessed by all users on the local computer.
- **File DSN tab** Enables you to add or remove a file data source from the local computer. File data sources can be shared by all users who have the same driver installed.
- **Drivers tab** Lists the installed ODBC drivers.
- **Tracing tab** Enables you to specify how the ODBC Driver Manager traces calls to ODBC functions. You can configure tracing separately for each installed ODBC application.
- **Connection Pooling tab** Enables you to select connection options for each installed driver.
- **About tab** Lists the installed ODBC component files.

After you add a data source, you can use the ODBC Data Source Administrator dialog box to configure the access to your data source. Select a data source, and then click one of the tabs to edit or review the information.

Connecting to a Data Source (ODBC Driver for Oracle)

**Important**

An ODBC application can pass connection information in a number of ways. For example, the application might have the driver always prompt the user for connection information. Or the application might expect a connection string that specifies the data source connection. How you connect to a data source depends on the connection method used by your ODBC application.

One common way to connect to a data source is through the Data Source dialog box. If your ODBC application is set up to use a dialog box, that dialog box is displayed and prompts you for the appropriate data source connection information.

You can also connect to a data source using the connection string.

To connect to a data source using a dialog box

1. When the Data Source dialog box appears, select an Oracle data source and then click OK. The Connect dialog box appears.
2. Fill in the appropriate information for the Connect dialog box, and then click OK.

After the connection information is verified, your application can use the ODBC Driver for Oracle to access the information that the data source contains.

© 2016 Microsoft

Determining Installed Oracle Components

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

To determine the Oracle components installed on your system (and their versions), navigate to the \Orainst directory under the Oracle home directory. Open one of the following text files: Nt.rgs, Win95.rgs, or Win98.rgs.

The file format is similar to the following:

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntinstall</td>
<td>3.3.1.0.8C</td>
<td>Oracle Installer</td>
</tr>
<tr>
<td>w32tcp80</td>
<td>8.0.5.0.0</td>
<td>Oracle TCP/IP Pro</td>
</tr>
<tr>
<td>w32nmp80</td>
<td>8.0.5.0.0</td>
<td>Oracle Named Pipe</td>
</tr>
<tr>
<td>w32utl80</td>
<td>8.0.5.0.0</td>
<td>Oracle Utilities</td>
</tr>
<tr>
<td>w32rsf80</td>
<td>8.0.5.0.0</td>
<td>Required Support</td>
</tr>
<tr>
<td>w32netclt80</td>
<td>8.0.5.0.0</td>
<td>Oracle Net8 Client</td>
</tr>
<tr>
<td>w32plus80</td>
<td>8.0.5.0.0</td>
<td>SQL*Plus</td>
</tr>
</tbody>
</table>

The .rgs files also include installation information and descriptions of each component.

© 2016 Microsoft

Setting the Date Format on Connection

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The new version of the Microsoft ODBC Driver for Oracle does not automatically set the date format for Oracle date fields. Previously when the driver connected, it used `ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH:MI:SS'`.

To set the date format, call `ALTER SESSION SET` and then perform the insert. For example:

```
conn.Execute "ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH:MI:SS'
ssSql = "INSERT INTO DATETEST VALUES (24,'1988-12-01 10:23:03')"
conn.Execute sSql
```

© 2016 Microsoft

ODBC Driver for Oracle Programmer's Reference

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

This section includes the following topics:

- Connection String Format and Attributes
- Connect Options
- Statement Options
- Cursor Type and Concurrency Combinations
- Using Microsoft Component Services
Connection String Format and Attributes

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

Instead of using a dialog box, some applications might require a connection string that specifies data source connection information. The connection string is made up of a number of attributes that specify how a driver connects to a data source. An attribute identifies a specific piece of information that the driver needs to know before it can make the appropriate data source connection. Each driver might have a different set of attributes, but the connection string format is always the same. A connection string has the following format:

```
"DSN=data-source-name[;SERVER=value] [;PWD=value] [;UID=value] [;<Attribute>=<value>]"
```

Note

The Microsoft ODBC Driver for Oracle supports the connection string format of the first version of the driver, which used CONNECTSTRING= instead of SERVER=.

If you are connecting to a data source provider that supports Windows authentication, you should specify Trusted_Connection=yes instead of user ID and password information in the connection string.

You must specify the data source name if you do not specify the UID, PWD, SERVER (or CONNECTSTRING), and DRIVER attributes. However, all other attributes are optional. If you do not specify an attribute, that attribute defaults to the one specified in the relevant DSN tab of the ODBC Data Source Administrator dialog box. The attribute value might be case-sensitive.

The attributes for the connection string are as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN</td>
<td>The data source name listed in the Drivers tab of the ODBC Data Source Administrator dialog box.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>PWD</td>
<td>The password for the Oracle Server that you want to access. This driver supports limitations that Oracle places on passwords.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>SERVER</td>
<td>The connect string for the Oracle Server that you want to access.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>UID</td>
<td>The Oracle Server user name. Depending on your system, this attribute might not be optional — that is, certain databases and tables might require this attribute for security purposes. Use &quot;/&quot; to use Oracle's operating system authentication.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>BUFFERSIZE</td>
<td>The optimal buffer size used when fetching columns. The driver optimizes fetching so that one fetch from the Oracle Server returns enough rows to fill a buffer of this size. Larger values tend to increase performance if you fetch a lot of data.</td>
<td>65535</td>
</tr>
<tr>
<td>SYNONYMCOLUMNS</td>
<td>When this value is true (1), an SQLColumn() API call returns column information. Otherwise, SQLColumn() returns only columns for tables and views. The ODBC Driver for Oracle provides faster access when this value is not set.</td>
<td>1</td>
</tr>
<tr>
<td>REMARKS</td>
<td>When this value is true (1), the driver returns Remarks columns for the SQLColumns result set. The ODBC Driver for Oracle provides faster access when this value is not set.</td>
<td>0</td>
</tr>
</tbody>
</table>
Oracle provides faster access when this value is not set.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>StdDayOfWeek</td>
<td>Enforces the ODBC standard for the DAYOFWEEK scalar. By default this is turned on, but users who need the localized version can change the behavior to use whatever Oracle returns.</td>
<td>1</td>
</tr>
<tr>
<td>GuessTheColDef</td>
<td>Specifies whether or not the driver should return a non-zero value for the cbColDef argument of SQLDescribeCol. Applies only to columns where there is no Oracle-defined scale, such as computed numeric columns and columns defined as NUMBER without a precision or scale. A SQLDescribeCol call returns 130 for the precision when Oracle does not provide that information.</td>
<td>0</td>
</tr>
</tbody>
</table>

For example, a connection string that connects to the MyDataSource data source using the MyOracleServerOracle Server and the Oracle User MyUserID would be:

```
"DSN={MyDataSource};UID={MyUserID};PWD={MyPassword};SERVER={MyOracleServer}"
```

A connection string that connects to the MyOtherDataSource data source using operating system authentication and the MyOtherOracleServerOracle Server would be:

```
"DSN=MyOtherDataSource;UID=/;PWD=;SERVER=MyOtherOracleServer"
```

© 2016 Microsoft

## Connect Options

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

These options allow customization of the database connection within an application.

<table>
<thead>
<tr>
<th>Connect option</th>
<th>Notes</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_AUTOCOMMIT</td>
<td>If you choose SQL_AUTOCOMMIT_OFF, your application must explicitly commit or roll back transactions with SQLTransact.</td>
<td>SQL_AUTOCOMMIT_OFF, your application must explicitly commit or roll back transactions with SQLTransact.</td>
</tr>
<tr>
<td>SQL_ODBC_CURSORS</td>
<td>This connection attribute is implemented in the Driver Manager.</td>
<td>This connection attribute is implemented in the Driver Manager.</td>
</tr>
<tr>
<td>SQL_OPT_TRACE</td>
<td>This connection attribute is implemented in the Driver Manager.</td>
<td>This connection attribute is implemented in the Driver Manager.</td>
</tr>
<tr>
<td>SQL_OPT_TRACEFILE</td>
<td>This connection attribute is implemented in the Driver Manager.</td>
<td>This connection attribute is implemented in the Driver Manager.</td>
</tr>
<tr>
<td>SQL_TRANSLATE_DLL</td>
<td>Returns error: &quot;Driver not capable.&quot;</td>
<td>Returns error: &quot;Driver not capable.&quot;</td>
</tr>
<tr>
<td>SQL_TRANSLATE_OPTION</td>
<td>A 32-bit value passed to the translation .dll.</td>
<td>A 32-bit value passed to the translation .dll.</td>
</tr>
<tr>
<td>SQL_TXN_ISOLATION</td>
<td>The driver allows only SQL_TXN_READ_COMMITTED. The following vParams are not supported: SQL_TXN_READ_UNCOMMITTED, SQL_TXN_REAPEATABLE_READ, SQL_TXN_SERIALIZABLE</td>
<td>The driver allows only SQL_TXN_READ_COMMITTED.</td>
</tr>
<tr>
<td>SQL_ATTR_ENLIST_IN_DTC</td>
<td>This ODBC 3.0 connection attribute allows you to use the ODBC Driver for Oracle in distributed transactions coordinated by Microsoft Component Services (or MTS, if you are using Windows NT). It provides the interface pointer pITransaction to the transaction as the vParam argument.</td>
<td>This ODBC 3.0 connection attribute allows you to use the ODBC Driver for Oracle in distributed transactions coordinated by Microsoft Component Services (or MTS, if you are using Windows NT). It provides the interface pointer pITransaction to the transaction as the vParam argument.</td>
</tr>
<tr>
<td>SQL_ATTR_CONNECTION_DEAD</td>
<td>This read-only ODBC 3.5 connection attribute allows you to determine whether the connection to the Oracle server has failed. Get only; cannot Set.</td>
<td>This read-only ODBC 3.5 connection attribute allows you to determine whether the connection to the Oracle server has failed. Get only; cannot Set.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

## Statement Options
These options allow customization of a specific execution statement within an application.

<table>
<thead>
<tr>
<th>Statement option</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_BIND_TYPE</td>
<td>Cannot exceed 2,147,483,647 bytes or available memory.</td>
</tr>
<tr>
<td>SQL_CONCURRENCY</td>
<td>For allowed values, see the Cursor Type and Concurrency Combinations.</td>
</tr>
<tr>
<td>SQL_CURSOR_TYPE</td>
<td>The driver does not allow SQL_CURSOR_DYNAMIC. See SQLSetScrollOptions for more information. For allowed values, see the Cursor Type and Concurrency Combinations.</td>
</tr>
<tr>
<td>SQL_GET_BOOKMARK</td>
<td>Returns a 32-bit integer value that is the bookmark for the current record number. Get only; cannot Set.</td>
</tr>
<tr>
<td>SQL_KEYSET_SIZE</td>
<td>Can be set only to 0.</td>
</tr>
<tr>
<td>SQL_MAX_ROWS</td>
<td>The maximum number of rows to return from a result set.</td>
</tr>
<tr>
<td>SQL_ROW_NUMBER</td>
<td>Returns a 32-bit integer specifying the position of the current row within the result set. Get only; cannot Set.</td>
</tr>
<tr>
<td>SQL_ROWSET_SIZE</td>
<td>Cannot exceed 4,294,967,296 rows; however, you must have enough virtual memory in your computer to handle your request.</td>
</tr>
<tr>
<td>SQL_USE_BOOKMARKS</td>
<td>Supports setting SQL_USE_BOOKMARKS to SQL_UB_ON and exposes fixed-length bookmarks.</td>
</tr>
</tbody>
</table>

Cursor Type and Concurrency Combinations

<table>
<thead>
<tr>
<th>Cursor type</th>
<th>Concurrency (allowed values)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CURSOR_FORWARD_ONLY</td>
<td>SQL_CONCUR_READ_ONLY</td>
</tr>
<tr>
<td>SQL_CURSOR_STATIC</td>
<td>SQL_CONCUR_READ_ONLY</td>
</tr>
</tbody>
</table>


[2] SQL_CONCUR_LOCK is supported only when the SQL_AUTOCOMMIT connection option is set to SQL_AUTOCOMMIT_OFF.

See Also

Connect Options

Using Microsoft Component Services

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.
You can enable an Oracle database to work with transactional Component Services (or MTS, if you are using Windows NT) on Microsoft Windows NT/Windows 2000 and Microsoft Windows 95/98. To enable an Oracle database to work with Component Services that support transactions, system administrators should create a view named VSXATRANS$. To create this script, you must run an Oracle-supplied script. For more information, see the Component Services Help or your Oracle documentation.

---

**Using Microsoft Internet Information Services**

- **Important**

  This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

  If you have difficulty connecting from within an IIS script (particularly if you receive an ORA-12641 error), add the following line to the Sqlnet.ora file:

  ```
  SQLNET.AUTHENTICATION_SERVICES = (none)
  ```

  This will disable the Secure Network Services so you can connect using anonymous authentication.

---

**Using Operating System Authentication**

- **Important**

  This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

  Oracle operating system authentication relies on the underlying operating system to control access to database accounts. Users need not enter a password when using this type of login.

  To take advantage of this feature, specify "/" as the user ID and do not specify a password when connecting using any of the following connection APIs: SQLBrowseConnect, SQLConnect, or SQLDriverConnect.

  Oracle databases use SQL*Net Authentication Services to authenticate users that are logged on. This service works well if users are logged into Oracle through SQLPlus; however, when the logged-in user is a service such as Internet Information Services, the authentication fails. This is a known limitation of SQL*Net Authentication and produces the following error: "[Microsoft][ODBC driver for Oracle][Oracle]ORA-12641: TNS: authentication service failed to initialize."

  You can correct this problem by editing the Sqlnet.ora file. This configuration file is usually stored in the Network\Admin subdirectory of the Oracle home directory. Add the following line to Sqlnet.ora:

  ```
  SQLNET.AUTHENTICATION_SERVICES = (none)
  ```

---

**Using SQLConfigDatasource with the ODBC Driver for Oracle**

- **Important**

  This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

  The following table lists valid SQLConfigDatasource settings for the Microsoft ODBC Driver for Oracle, version 1.0 (Msorc10.dll) and the Microsoft ODBC Driver for Oracle, version 2.0 (Msorc32.dll):

<table>
<thead>
<tr>
<th>SQLConfigDatasource Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLNET.AUTHENTICATION_SERVICES</td>
</tr>
<tr>
<td>SQLNET.AUTHENTICATION_SERVICES</td>
</tr>
</tbody>
</table>

  Note

  The Msorc10.dll driver (version 1.0) supports all settings except Server. The Msorc32.dll driver (version 2.0 and higher) supports all settings.

  Some settings are ignored by the driver but are accepted by SQLConfigDatasource. Including these settings in the ODBC connection string is the only way they will be accepted at run time. An ignored setting will not be stored in the registry when SQLConfigDatasource creates the data source.
In the following table, A/N means any valid alphanumeric string up to the maximum allowable length. Max Len (maximum length) is the maximum allowable string length accepted by the setting, including the string-terminator character.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Max Len</th>
<th>Default value</th>
<th>Valid values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferSize</td>
<td>7</td>
<td>65535</td>
<td>1000</td>
<td>Minimum fetch buffer size up to 65535 bytes</td>
</tr>
<tr>
<td>CatalogCap</td>
<td>2</td>
<td>1</td>
<td>0 or 1</td>
<td>If 1, nonquoted identifiers will be converted to uppercase in the catalog functions.</td>
</tr>
<tr>
<td>ConnectString</td>
<td>128</td>
<td>A/N</td>
<td></td>
<td>Connection string. Required method of specifying the server name with the Msoorc10.dll driver.</td>
</tr>
<tr>
<td>Description</td>
<td>256</td>
<td>A/N</td>
<td></td>
<td>Description.</td>
</tr>
<tr>
<td>DSN</td>
<td>33</td>
<td>A/N</td>
<td></td>
<td>Data source name.</td>
</tr>
<tr>
<td>GuessTheColDef</td>
<td>4</td>
<td>0</td>
<td>A/N</td>
<td>Returns a non-zero value for columns without Oracle-defined scale.</td>
</tr>
<tr>
<td>NumberFloat</td>
<td>2</td>
<td>0 or 1</td>
<td></td>
<td>If 0, FLOAT columns are treated as SQL_FLOAT. If 1, FLOAT columns are treated as SQL_DOUBLE.</td>
</tr>
<tr>
<td>PWD</td>
<td>30</td>
<td>A/N</td>
<td></td>
<td>Password.</td>
</tr>
<tr>
<td>RDOSupport</td>
<td>2</td>
<td>0 or 1</td>
<td></td>
<td>Allows RDO to call Oracle procedures.</td>
</tr>
<tr>
<td>Remarks</td>
<td>2</td>
<td>0</td>
<td>0 or 1</td>
<td>Include REMARKS in catalog functions.</td>
</tr>
<tr>
<td>RowLimit</td>
<td>4</td>
<td>0 to 99</td>
<td></td>
<td>Maximum number of rows returned by a SELECT statement. A zero-length string indicates that no limit is applied.</td>
</tr>
<tr>
<td>Server</td>
<td>128</td>
<td>A/N</td>
<td></td>
<td>Oracle server name.</td>
</tr>
<tr>
<td>SynonymColumns</td>
<td>2</td>
<td>1</td>
<td>0 or 1</td>
<td>Include SYNONYMs in SQLColumns.</td>
</tr>
<tr>
<td>SystemTable</td>
<td>2</td>
<td>0 or 1</td>
<td></td>
<td>If 0, system tables will not be displayed. If 1, system tables will be displayed.</td>
</tr>
<tr>
<td>TranslationDLL</td>
<td>33</td>
<td>A/N</td>
<td></td>
<td>Translation .dll name.</td>
</tr>
<tr>
<td>TranslationName</td>
<td>33</td>
<td>A/N</td>
<td></td>
<td>Translation name.</td>
</tr>
<tr>
<td>TranslationOption</td>
<td>33</td>
<td>A/N</td>
<td></td>
<td>Translation option.</td>
</tr>
<tr>
<td>TxnCap</td>
<td>2</td>
<td>A/N</td>
<td></td>
<td>Transaction capable. If 0, the driver reports that it does not support transactions. If 1, the driver reports that it is capable of performing transactions.</td>
</tr>
<tr>
<td>UID</td>
<td>30</td>
<td>A/N</td>
<td></td>
<td>User name.</td>
</tr>
</tbody>
</table>

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

You must be able to retrieve a single ROWID column for the table queried. A keyset-driven cursor cannot be used on joins, queries, or statements that contain DISTINCT, GROUP BY, UNION, INTERSECT, or MINUS clauses.

Also, if your application uses table aliases, keyset-driven cursors will not work; forward-only or static cursor types are required. Using the keyset cursor type with table aliases will cause the following error: "[Microsoft][ODBC driver for Oracle]Cannot use Keyset-driven cursor on join, with union, intersect or minus or on read only result set."

Note

Because of the way the driver handles the SQL statement that is sent to the Oracle server, Oracle internally returns the following error message: "ORA-00964: table name not in FROM list."
Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

This section includes the following topics:

- Stored Procedure Parameter Limitations
- Using Synonyms with Stored Procedures
- Revoking and Granting Rights When Using Stored Procedures
- Returning Array Parameters from Stored Procedures

© 2016 Microsoft

Stored Procedure Parameter Limitations

When running Oracle stored procedures that utilize 10 or more output parameters, the stored procedure call will fail, resulting in an Access Violation or ActiveX Data Objects (ADO) error. This can occur when using the Microsoft ODBC Driver for Oracle with versions 8.0.4.0.0 and 8.0.4.0.4 of the Oracle client software.

To correct the problem, the Oracle client software must be upgraded to version 8.0.4.2.0 or higher. Contact Oracle Corporation for more information about the patches.

Note

This problem does not occur with the early release of Oracle client software version 8.0.3.0.0.

© 2016 Microsoft

Using Synonyms with Stored Procedures

The Microsoft ODBC Driver for Oracle versions 2.0 and 2.5 do not support synonyms when calling Oracle stored procedures. Synonyms work as expected when used with other Oracle database objects such as tables.

© 2016 Microsoft

Revoking and Granting Rights When Using Stored Procedures

The Microsoft ODBC Driver for Oracle returns the following error message when user rights are granted and then revoked on a table accessed by a stored procedure:

SQL_ERROR=-1
szErrorMsg="[Microsoft][ODBC driver for Oracle]Wrong number of parameters"
szErrorMsg="[Microsoft][ODBC driver for Oracle]Syntax error or access violation"
The call to the Oracle OCI function Odessp() fails in this scenario but is necessary in order to implement default parameters. After the underlying table permissions are modified, the stored procedure must be recompiled before running it again.

Returning Array Parameters from Stored Procedures

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

In Oracle 7.3, there is no way to access a PL/SQL Record Type except from a PL/SQL program. If a packaged procedure or function has a formal argument defined as a PL/SQL Record Type, it is not possible to bind that formal argument as a parameter. Use the PL/SQL TABLE type in the Microsoft ODBC Driver for Oracle to invoke array parameters from procedures containing the correct escape sequences.

To invoke the procedure, use the following syntax:

```
{call <package-name>.<proc-or-func>;
 (..., <resultset <max-records-requested>, <formal-array-param_1>,
 <formal-array-param_2>,..., <formal-array-param_n>, ... ) }
```

Note

The <max-records-requested> parameter must be greater than or equal to the number of rows present in the result set. Otherwise, Oracle returns an error that is passed to the user by the driver.

PL/SQL records cannot be used as array parameters. Each array parameter can represent only one column of a database table.

The following example defines a package containing two procedures that return different result sets, and then provides two ways to return result sets from the package.

Package definition:

```sql
CREATE OR REPLACE PACKAGE SimplePackage AS

TYPE t_id is TABLE OF NUMBER(5)
  INDEX BY BINARY_INTEGER;

TYPE t_Course is TABLE OF VARCHAR2(10)
  INDEX BY BINARY_INTEGER;

TYPE t_Dept is TABLE OF VARCHAR2(5)
  INDEX BY BINARY_INTEGER;

PROCEDURE proc1
  (o_id OUT t_id,
   ao_course OUT t_Course,
   ao_dept OUT t_Dept);

PROCEDURE proc2
  (i_Arg1 IN NUMBER,
   ao_Arg2 OUT t_pk1Type1,
   ao_Arg3 OUT t_pk1Type2);

END SimplePackage;

CREATE OR REPLACE PACKAGE BODY SimplePackage AS

PROCEDURE proc1 (o_id OUT t_id,
  ao_course OUT t_Course, ao_dept OUT t_Dept ) AS
BEGIN
  o_id(1) := 200;
  ao_course(1) := 'M101';
  ao_dept(1) := 'EEE';

  o_id(2) := 201;
  ao_course(2) := 'PHY320';
  ao_dept(2) := 'ECE';
END proc1;
```
To invoke procedure PROC1

1. Return all the columns in a single result set:

   (call SimplePackage.Proc1( {resultset 3, o_id, ao_course, ao_dept } ))

2. Return each column as a single result set:

   (call SimplePackage.Proc1( {resultset 3, o_id}, {resultset 3, ao_course}, {resultset 3, ao_dept} ))

This returns three result sets, one for each column.

To invoke procedure PROC2

1. Return all the columns in a single result set:

   (call SimplePackage.Proc2( 5, (resultset 5, ao_Arg2, ao_Arg3) ))

2. Return each column as a single result set:

   (call SimplePackage.Proc2( 5, (resultset 5, ao_Arg2), (resultset 5, ao_Arg3) ))

Ensure that your applications fetch all the result sets using the SQLMoreResults API. For more information, refer to the ODBC Programmer’s Reference.

Note

In the ODBC Driver for Oracle version 2.0, Oracle functions that return PL/SQL arrays cannot be used to return result sets.

© 2016 Microsoft

Error Messages (ODBC Driver for Oracle)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

When an error occurs, the Microsoft ODBC Driver for Oracle returns the SQLSTATE (an ODBC error code) and an error message. The driver derives this information both from errors detected by the driver and from errors returned by the Oracle Server.

This section includes the following topic:

- Messages Returned by the ODBC Driver for Oracle
Messages Returned by the ODBC Driver for Oracle

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

If an Oracle error message is available, it will be returned preceded by the [Microsoft], [ODBC Driver for Oracle], and [Oracle] tags; otherwise, the message is returned without the [Oracle] tag as in the following examples:

Oracle error message:

[Microsoft][ODBC driver for Oracle][Oracle]ORA-nnnnn message-text

ODBC Driver for Oracle error message:

[Microsoft][ODBC driver for Oracle]

ODBC Driver for Oracle Conformance Levels

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

ODBC defines two types of conformance standards for drivers: the API conformance standard and the SQL grammar conformance standard. API conformance refers to the functions that a driver supports. SQL conformance refers to the SQL grammar that the driver supports. Each conformance standard is made up of levels.

This section provides conformance information about the ODBC Driver for Oracle.

This section contains the following topics:

- API Conformance Level (ODCB Driver for Oracle)
- Supported Options (ODCB Driver for Oracle)
- SQL Conformance Levels (ODCB Driver for Oracle)

API Conformance Level (ODCB Driver for Oracle)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The ODBC Driver for Oracle supports the Core Level API functions and Level 1 API functions. The driver also supports the following Level 2 API functions:

- SQLBrowseConnect( )
- SQIDataSources( )
- SQLDescribeParam( )
Supported Options (ODBC Driver for Oracle)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The ODBC Driver for Oracle supports the following options for the SQLGetConnectOption() and SQLSetConnectOption() Level 1 functions:

- SQL_ACCESS_MODE (SQLGetConnectOption() only)
- SQL_AUTOCOMMIT
- SQL_ODBC_CURSOR
- SQL_OPT_TRACEFILE
- SQL_OPT_TRACE
- SQL_TRANSLATE_DLL
- SQL_TRANSLATE_OPTION
- SQL_TXN_ISOLATION

The ODBC Driver for Oracle supports the following options for the SQLGetStmtOption() and SQLSetStmtOption() Level 1 functions:

- SQL_BINO_TYPE
- SQL_CONCURRENCY
- SQL_CURSOR_TYPE
- SQL_KEYSET_SIZE
- SQL_MAX_ROWS
- SQL_ROWSET_SIZE

SQL Conformance Levels (ODBC Driver for Oracle)

Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The ODBC Driver for Oracle supports the Minimum SQL grammar and Core SQL grammar and also supports the following ODBC extensions to SQL:

- Date, time, and timestamp data
- Left and right outer joins
- Numeric functions:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs</td>
<td>Log</td>
<td>round</td>
<td>tan</td>
</tr>
<tr>
<td>Ceiling</td>
<td>Log10</td>
<td>second</td>
<td>truncate</td>
</tr>
<tr>
<td>Cos</td>
<td>Mod</td>
<td>sign</td>
<td></td>
</tr>
<tr>
<td>Exp</td>
<td>Pi</td>
<td>sin</td>
<td></td>
</tr>
<tr>
<td>Floor</td>
<td>Power</td>
<td>sqrt</td>
<td></td>
</tr>
</tbody>
</table>

- Date functions:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Curdate</td>
<td>Dayofweek</td>
<td>monthname</td>
<td>second</td>
</tr>
<tr>
<td>Curtime</td>
<td>Dayofyear</td>
<td>minute</td>
<td>week</td>
</tr>
<tr>
<td>Dayname</td>
<td>Hour</td>
<td>now</td>
<td>year</td>
</tr>
<tr>
<td>Dayofmonth</td>
<td>Month</td>
<td>quarter</td>
<td></td>
</tr>
</tbody>
</table>

- String functions:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ascii</td>
<td>Left</td>
<td>right</td>
<td>ucase</td>
</tr>
<tr>
<td>Char</td>
<td>Length</td>
<td>rtrim</td>
<td></td>
</tr>
<tr>
<td>Concat</td>
<td>Ltrim</td>
<td>soundex</td>
<td></td>
</tr>
<tr>
<td>Lcase</td>
<td>Replace</td>
<td>substring</td>
<td></td>
</tr>
</tbody>
</table>

- Type-conversion function:

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert</td>
</tr>
</tbody>
</table>

- System functions:

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifnull</td>
</tr>
<tr>
<td>User</td>
</tr>
</tbody>
</table>

API Functions (ODBC Driver for Oracle)

⚠️ Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The Microsoft ODBC Driver for Oracle supports the Core Level, Level 1, and Level 2 API functions. These functions are listed in ODBC Conformance Levels.

- **Core Level Interface** (CLI) conformance provides features defined in the ISO CLI specification and the mandatory features defined in the Open Group CLI specification.
- **Level 1** conformance provides Core Level Interface functionality as well as additional features such as transactions.
- **Level 2** conformance provides Level 1 functionality as well as additional features such as bookmarks, dynamic parameters, and asynchronous execution of ODBC functions.
# Thread-Safety Notes on API Functions (ODBC Driver for Oracle)

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The Microsoft ODBC Driver for Oracle is thread-safe; however, Oracle does not allow multiple concurrent statements on a single connection. The driver enforces this restriction. In other words, in multithreaded applications, although any thread can call into the ODBC Driver for Oracle at any time, the driver blocks any other thread from the driver on the same connection until the original thread leaves the driver.

The driver does not block if there are two statements on two different connections. However, if there is a single connection with two statements, there is potential for blocking.

# Core Level API Functions (ODBC Driver for Oracle)

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

Functions at this level comprise the minimum level of interface conformance for ODBC drivers.

<table>
<thead>
<tr>
<th>API function</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocConnect</td>
<td>Allocates memory for a connection handle, hdbc, within the environment identified by henv. The Driver Manager processes this call and calls the driver’s SQLAllocConnect function whenever SQLConnect, SQLBrowseConnect, or SQLDriverConnect is called.</td>
</tr>
<tr>
<td>SQLAllocEnv</td>
<td>Displays a dialog box specifying the requirement for Oracle Client software and then returns SQL_NULL_HANDLE. If the Oracle Client software is not installed, this function allocates memory for an environment handle, henv, and initializes the ODBC call-level interface for use by an application.</td>
</tr>
<tr>
<td>SQLAllocStmt</td>
<td>Allocates memory for a statement handle and associates the statement handle with the connection specified by hdbc. The Driver Manager passes this call to the driver, which allocates the memory for the hstmt structure.</td>
</tr>
<tr>
<td>SQLBindCol</td>
<td>Assigns storage space for a result column and specifies the type of the result.</td>
</tr>
<tr>
<td>SQLCancel</td>
<td>Cancels the processing on a statement handle, hstmt. In some cases, Oracle does not allow cancellation of a running statement. This means that a running statement will continue until Oracle completes the process, at which time the results from the statements are canceled by the ODBC Driver for Oracle.</td>
</tr>
<tr>
<td>SQLColAttributes</td>
<td>Returns descriptor information for a column in a result set. Descriptor information is returned as a character string, a 32-bit descriptor-dependent value, or an integer value.</td>
</tr>
<tr>
<td>SQLConnect</td>
<td>Connects to a data source. To use Oracle Operating System Authentication, specify &quot;/*&quot; as the szUID parameter and &quot;&quot; as the szAuthStr parameter.</td>
</tr>
<tr>
<td>SQLDescribeCol</td>
<td>Returns the name, type, precision, scale, and nullability of the given result column.</td>
</tr>
<tr>
<td>SQLDisconnect</td>
<td>Closes a connection. If connection pooling is enabled for a shared environment and an application calls SQLDisconnect on a connection in that environment, the connection is returned to the connection pool and is still available to other components using the same shared environment.</td>
</tr>
<tr>
<td>SQLError</td>
<td>Returns error or status information about the last error. The driver maintains a stack or list of errors that can be returned for the hstmt, hdbc, and henv arguments, depending on how the call to SQLError is made. The error queue is flushed after each statement. Usually retrieves an Oracle error message and is otherwise empty.</td>
</tr>
<tr>
<td>SQLExecDirect</td>
<td>Executes a new, unprepared SQL statement. The driver uses the current values of the parameter marker variables if any parameters exist in the statement. If your table, view, or field names contain spaces, enclose the names in back quote marks. For example, if your database contains a table named My Table and the field My Field, enclose each element of the identifier like so:</td>
</tr>
</tbody>
</table>
### Level 1 API Functions (ODBC Driver for Oracle)

<table>
<thead>
<tr>
<th>API function</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLColumns</td>
<td>Creates a result set for a table, which is the column list for the specified table or tables. When you request columns for a PUBLIC synonym, you must have set the SYNONYM_COLUMNS connection attribute and specified an empty string as the szTableOwner argument. When returning columns for PUBLIC synonyms, the driver sets the TABLE NAME column to an empty string. The result set contains an additional column, ORDINAL POSITION, at the end of each row. This value is the ordinal position of the column in the table.</td>
</tr>
<tr>
<td>SQLDriverConnect</td>
<td>Connects to an existing data source. For details, see Connection String Format and Attributes.</td>
</tr>
<tr>
<td>SQLGetConnectOption</td>
<td>Returns the current setting of a connection option. This function is partially supported. The driver supports all values for the fOption argument but does not support some vParam values for the fOption argument SQL_TXN_ISOLATION. For more information, see Connect Options.</td>
</tr>
<tr>
<td>SQLGetData</td>
<td>Retrieves the value of a single field in the current record of the given result set.</td>
</tr>
<tr>
<td>SQLGetFunctions</td>
<td>Returns TRUE for all supported functions. Implemented by the Driver Manager.</td>
</tr>
<tr>
<td>SQLGetInfo</td>
<td>Returns information, including SQLDBC, SQLSMALLINT, SQLPOINTER, SQLSMALLINT, and SQLSMALLINT *, about the ODBC Driver for Oracle and data source associated with a connection handle, hdbc.</td>
</tr>
<tr>
<td>API function</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>SQLBindParameter</td>
<td>Associates a buffer with a parameter marker in an SQL statement.</td>
</tr>
<tr>
<td>SQLBrowseConnect</td>
<td>Returns successive levels of attributes and attribute values.</td>
</tr>
<tr>
<td>SQLDataSources</td>
<td>Lists data source names. Implemented by the Driver Manager.</td>
</tr>
<tr>
<td>SQLDescribeParam</td>
<td>Returns the description of a parameter marker associated with a prepared SQL statement. Returns a best guess of what the parameter is, based on parsing the statement. If the parameter type cannot be determined, SQL_VARCHAR returns with length 2000.</td>
</tr>
<tr>
<td>SQLDrivers</td>
<td>Implemented by the Driver Manager.</td>
</tr>
<tr>
<td>SQLExtendedFetch</td>
<td>Similar to SQLFetch but returns multiple rows using an array for each column. The result set is forward-scrollable and can be made backward-scrollable if the cursor is defined to be static, not forward-only. For forward-only cursors with default column binding, column data from data sets larger than the BUFFERSIZE connection attribute is fetched directly into data buffers. Does not support variable-length bookmarks and does not support fetching a rowset at an offset (other than 0) from a bookmark.</td>
</tr>
<tr>
<td>SQLForeignKeys</td>
<td>Returns a list of foreign keys in a single table, or a list of foreign keys in other tables that refer to a single table.</td>
</tr>
<tr>
<td>SQLMoreResults</td>
<td>Determines whether more results are pending on a statement handle, hstmt, containing SELECT, UPDATE, INSERT, or DELETE statements and if so, initializes processing for those results. Oracle supports multiple result sets only from stored procedures, when using {resultset... } escape sequences.</td>
</tr>
<tr>
<td>SQLNativeSql</td>
<td>For information about usage, see Returning Array Parameters from Stored Procedures.</td>
</tr>
<tr>
<td>SQLNumParams</td>
<td>Returns the number of parameters in an SQL statement. The number of parameters should equal the number of question marks in the SQL statement passed to SQLPrepare.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

Level 2 API Functions (ODBC Driver for Oracle)

⚠️ Important

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

Functions at this level provide Level 1 interface conformance plus additional functionality such as support for bookmarks, dynamic parameters, and asynchronous execution of ODBC functions.
### SQLPrimaryKeys

Returns the column names that comprise the primary key for a table.

### SQLProcedureColumns

Returns a list of input and output parameters, the return value, the columns in the result set of a single procedure, and two additional columns, OVERLOAD and ORDINAL_POSITION. OVERLOAD is the OVERLOAD column from the ALL_ARGUMENTS table of the Oracle Data Dictionary View. ORDINAL_POSITION is the SEQUENCE column from the ALL_ARGUMENTS table of the Oracle Data Dictionary View. For packed procedures, the PROCEDURE NAME column is in packagename.procedurename format. Does not return the procedure columns of a created synonym that refers to a procedure or function.

### SQLProcedures

Returns a list of procedures in the data source. For packed procedures, the PROCEDURE NAME column is in packagename.procedurename format.

Because Oracle does not provide a way to distinguish packed procedures from packed functions, the driver returns SQL_PT_UNKNOWN for the PROCEDURE_TYPE column.

### SQLSetPos

Sets the cursor position in a rowset. You can use SQLSetPos with SQLGetData to retrieve rows from unbound columns after positioning the cursor to a specific row in the rowset. Rows added to the result set using fOption SQL_ADD are added after the last row in the result set.

### SQLSetScrollOptions

Sets options that control the behavior of cursors associated with a statement handle, hstmt. For details, see [Cursor Type and Concurrency Combinations](#).

---

**Supported Data Types (ODBC Driver for Oracle)**

---

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The ODBC Driver for Oracle supports all Oracle 7.3 data types; however, it does not support any of the new Oracle8 data types listed here.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Oracle 7.3</th>
<th>Oracle8</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFILE</td>
<td>n/a</td>
<td>Not supported</td>
</tr>
<tr>
<td>BLOB</td>
<td>n/a</td>
<td>Not supported</td>
</tr>
<tr>
<td>CHAR</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>CLOB</td>
<td>n/a</td>
<td>Not supported</td>
</tr>
<tr>
<td>DATE</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>LONG</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>NCHAR</td>
<td>n/a</td>
<td>Not supported</td>
</tr>
<tr>
<td>NCLOB</td>
<td>n/a</td>
<td>Not supported</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>NVARCHAR2</td>
<td>n/a</td>
<td>Not supported</td>
</tr>
<tr>
<td>RAW</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>MLSLABEL</td>
<td>Not supported.</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>

**Note**

For more information about the allowable size of the VARCHAR column, see [VARCHAR Column Size](#) in this guide.
Mapping Data Types (ODBC Driver for Oracle)

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

The Oracle Server supports a set of data types. The ODBC Driver for Oracle maps these data types to their appropriate ODBC SQL data types. The following table lists the Oracle 7.3 Server data types and their corresponding ODBC SQL data types.

The ODBC Driver for Oracle supports Oracle 7.3 and some Oracle8 data types. For more information about supported Oracle8 data types, see [Supported Data Types](#).

<table>
<thead>
<tr>
<th>Oracle Server data type</th>
<th>ODBC SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>SQL_CHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>SQL_TIMESTAMP</td>
</tr>
<tr>
<td>FLOAT</td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td>LONG</td>
<td>SQL_LONGVARCHAR</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>SQL_LONGVARBINARY</td>
</tr>
<tr>
<td>NUMBER</td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td>RAW</td>
<td>SQL_VARBINARY</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>SQL_VARCHAR</td>
</tr>
</tbody>
</table>

**Note**

For more information about the allowable size of the VARCHAR column, see [VARCHAR Column Size](#) in this guide.

VARCHAR Column Size (ODBC Driver for Oracle)

**Important**

This feature will be removed in a future version of Windows. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Instead, use the ODBC driver provided by Oracle.

In Oracle8, the maximum size of a VARCHAR column has increased from 2000 to 4000 bytes. The Oracle 7.3.x client software has no way to bind a parameter value larger than 2000 bytes. Therefore, if you create a table with a VARCHAR column of larger than 2000 bytes, you will be unable to perform parameterized inserts, updates, deletes, and queries against it with data that exceeds the 2000-byte limit of the client software. Because both the ODBC Driver for Oracle and the OLE DB Provider for Oracle use parameterized inserts, updates, deletes, and queries, they will report ORA-01026 errors in this case. Data that is within the limits enforced by the Oracle client software will work. To avoid this 2000-byte limit, you must upgrade your client software to Oracle8 (8.0.4.1.1c or higher).

Visual FoxPro ODBC Driver

Microsoft Visual FoxPro is a powerful object-oriented environment for database construction and application development. The Microsoft Visual FoxPro ODBC Driver enables applications to open, query, and update data in Visual FoxPro and earlier versions of FoxPro through the Open Database Connectivity (ODBC) interface.

For example, with the Microsoft Visual FoxPro ODBC Driver you can do the following:

- Use Microsoft Query to query and update Visual FoxPro data from Microsoft Excel worksheets.
- Create mail-merge letters using Visual FoxPro data with Microsoft Word.
- Query and update Visual FoxPro views and tables from Microsoft Access.
- Use Visual FoxPro as the data store for Microsoft Visual Basic, Microsoft Visual C++, and C applications.

You can use the driver to accomplish many other tasks. The following table lists a few topics to help you get started.

<table>
<thead>
<tr>
<th>To</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find out more about using Visual FoxPro data with Microsoft Office</td>
<td>Accessing Visual FoxPro Data from Microsoft Office</td>
</tr>
<tr>
<td>Learn about using Visual FoxPro data in Visual Basic applications</td>
<td>Using the Visual FoxPro ODBC Driver with Your Visual Basic Application</td>
</tr>
<tr>
<td>View a simple example using Visual C++ to access Visual FoxPro data</td>
<td>Using the Visual FoxPro ODBC Driver with Your C or C++ Application</td>
</tr>
<tr>
<td>See a list of supported hardware and software</td>
<td>System Requirements</td>
</tr>
</tbody>
</table>

For information about downloading and installing the Visual FoxPro ODBC driver, see Visual FoxPro ODBC Driver on MSDN Downloads.

© 2016 Microsoft

Driver Architecture Overview

The Microsoft Visual FoxPro ODBC Driver is a 32-bit driver that enables you to open and query a Microsoft Visual FoxPro database or FoxPro tables through the Open Database Connectivity (ODBC) interface. You can access FoxPro data using the following types of applications:

- A Microsoft Office application, such as Microsoft Excel or Microsoft Word, that uses Microsoft Query to communicate with ODBC.
- An application written in Microsoft Visual C++ or C that uses the ODBC SDK API.
- An application written in Microsoft Visual Basic or Microsoft Visual Basic for Applications.

In each case, the request for information uses the ODBC API. The ODBC Driver Manager works with the Visual FoxPro ODBC Driver to open and retrieve data from FoxPro tables and databases.

The architecture is represented in the following diagram:

This section contains the following topics.

- Visual FoxPro Terminology
- Installing and Configuring the Visual FoxPro ODBC Driver
- Using the Visual FoxPro ODBC Driver

© 2016 Microsoft

Visual FoxPro Terminology

database
   In Visual FoxPro, a database file has a .dbc extension and can contain one or more tables.

database table
   In Visual FoxPro, a table that is associated with a database. Contrast free table.

free table
   In Visual FoxPro, a table that is not associated with a database.

   A .dbf file created in FoxPro version 2.x is a free table unless it is converted to a Visual FoxPro table and added to a Visual FoxPro database. Contrast database table.

preparable SQL statement
   A SQL statement that has not already been processed by the SQLPrepare function. For more information about this function with the Visual FoxPro
table
In Visual FoxPro, records are stored in a table. Each row of a table represents a record, and the columns of the table represent the fields of the record. Each Visual FoxPro table is stored in its own file with a .dbf extension. Visual FoxPro tables can be associated with a database.

FoxPro version 2.x tables are not associated with a database.

Installing and Configuring

To download the Visual FoxPro ODBC Driver, see the Visual FoxPro ODBC Driver download site.

This section contains the following topics to help you install and configure the Microsoft Visual FoxPro ODBC Driver:

- System Requirements
- Supported Versions of FoxPro
- Setting Up the Visual FoxPro ODBC Driver
- ODBC Visual FoxPro Setup Dialog Box

To download the driver, see the Visual FoxPro ODBC Driver download site.

System Requirements (Visual FoxPro ODBC Driver)

The system requirements for installation provide the minimum operating system and disk space needed to successfully install the driver. After you've installed the driver, you can select the specific application software you want to use to access Visual FoxPro data.

Installation Requirements

For installation requirements and other information, see the Visual FoxPro ODBC Driver download site.

Accessing Visual FoxPro Data

To access Microsoft Visual FoxPro or FoxPro 2.x data, you must have the following:

- ODBC Client Software (automatically installed with the driver)
- Microsoft Visual FoxPro ODBC Driver
- Any of the following types of application software:
  - Microsoft Office application such as Microsoft Excel or Microsoft Word
  - C or C++ language ODBC application
  - Microsoft Visual Basic ODBC application
- Any of the following types of data:
  - Visual FoxPro database or a directory of free tables
  - FoxPro 2.0, 2.5, 2.6 table

The Visual FoxPro ODBC Driver supports double-byte character sets (DBCS). For more information, see International Support.

The driver does not support 16-bit Microsoft Windows 3.1 applications.

Supported Versions of FoxPro
You can use the Microsoft® Visual FoxPro® ODBC Driver to access data stored in FoxPro tables. The following versions of FoxPro data are supported:

- 2.0
- 2.5
- 2.6
- Visual FoxPro (all versions)

When you access data stored in Visual FoxPro, you can choose to connect to a database that contains zero or more tables or to a directory of free tables.

For more information about how to connect to a data source, see Adding a Visual FoxPro Data Source.

To download the driver, see the Visual FoxPro ODBC Driver download site.

© 2016 Microsoft

Setting Up the Visual FoxPro ODBC Driver

For information about downloading and installing the Visual FoxPro ODBC driver, see Visual FoxPro ODBC Driver on MSDN Downloads.

You use the Microsoft Visual FoxPro ODBC Driver Setup program to do the following:

- Add new components.
- Remove installed components.
- Reinstall to restore missing files and settings.
- Remove all previously installed components.

After you install the driver on your system, the Setup program recognizes the installed driver components and presents additional dialog boxes that enable you to change your driver’s configuration.

© 2016 Microsoft

ODBC Visual FoxPro Setup Dialog Box

The ODBC Visual FoxPro Setup dialog box enables you to add or change a Visual FoxPro data source.

To download the driver, see the Visual FoxPro ODBC Driver download site.

**Dialog Box Options**

**Data Source Name**
Type the name you want to use for the data source.

**Description**
Type a description for the data source.

**Database type**
Lets you choose the type of database you want your data source to connect to.

**Visual FoxPro database (.DBC)**
Specifies that the data source connects to a Visual FoxPro database (.dbc file) and to all the tables and local views in the database.

**Free Table directory**
Specifies that the data source connects to a directory of free tables. Any database tables in the same directory are ignored by ODBC catalog functions such as SQLColumns or SQLTables. Database tables can be accessed by using SQL SELECT statements sent through SQLExecute and SQLExecDirect.

**Path**
Displays the path and name for the database or the directory of free tables to which the data source connects.

**Browse**
Enables you to search your system and network for the database or directory to which you want to connect the data source.

**Options**
Expands the dialog box so that you can set Visual FoxPro ODBC Driver options.

**Driver**
Collating sequence

The sequence in which fields are sorted. The default sequences reflect the sequences supported by your language version of the operating system. For a list of supported collating sequences, see SET COLLATE.

Exclusive

When this check box is selected, the driver opens the Visual FoxPro database exclusively when you access data using the data source. Other users cannot access the database or the tables in the database while the database is opened exclusively. Tables within the exclusively opened database are opened as SHARED. To open a table exclusively, use the SET EXCLUSIVE command. This check box is disabled when Database type is set to Free Table directory.

Null

Determines whether columns created with ALTER TABLE and CREATE TABLE allow null values. If you set Null ON, INSERT – SQL inserts a null value into any column not included in an INSERT – SQL... VALUE clause. A blank is inserted if Null is OFF. You can also control this option through a passed connection string as in the following code:

```
strCon = "DRIVER=MICROSOFT VISUAL FOXPRO DRIVER;
SOURCETYPE=DBC;SOURCEDB=D:\Testdata.dbc;BACKGROUNDFETCH=NO;NULL=NO"
```

Deleted

Determines whether rows marked as deleted are returned. You can also control this option through a passed connection string as in the following code:

```
strCon = "DRIVER=MICROSOFT VISUAL FOXPRO DRIVER;
SOURCETYPE=DBC;SOURCEDB=D:\Testdata.dbc;BACKGROUNDFETCH=NO;
DELETED=YES"
```

Fetch data in background

Determines whether records will be fetched in the background (progressive fetching) or your application will wait until all records in the result set are fetched.

Using the Visual FoxPro ODBC Driver

In this section, the following topics discuss how to use the Microsoft® Visual FoxPro® ODBC Driver:

- Adding a Visual FoxPro Data Source
- Modifying a Visual FoxPro Data Source
- Deleting a Visual FoxPro Data Source
- Connecting to a Visual FoxPro Data Source
- Using Connection Strings
- Using the Visual FoxPro ODBC Driver with Your Visual Basic Application
- Using the Visual FoxPro ODBC Driver with Your C or C++ Application
- Accessing Visual FoxPro Data from Microsoft Office

Adding a Visual FoxPro Data Source

To access Visual FoxPro data from your application, you must have a data source. You can create a data source as follows:

- In an application, such as Microsoft® Word, Microsoft Excel, or Microsoft Access, that uses ODBC drivers.
- Outside your application, using the Microsoft Windows® 95, Microsoft Windows 98, or Microsoft Windows NT®/Windows 2000 Control Panel.

After a data source exists on your system, you can reuse the same data source every time that you want to access Visual FoxPro data. If you have several different databases or tables you want to access, you can create a separate data source for each database or directory.

The following procedure creates a data source by using Control Panel. For more information about how to create a data source from an application, see Accessing Visual FoxPro Data from Microsoft Office.
To add a Visual FoxPro data source

1. On computers that are running Windows 2000, open the Windows Control panel and double-click Administrative Tools.

2. Double-click Data Sources (ODBC) to open the ODBC Data Source Administrator dialog box. This icon is available after you have installed the Visual FoxPro ODBC Driver or any ODBC driver software.

   □ Note
   
   If you are running an earlier version of Windows, open the Windows Control panel and double-click 32-bit ODBC or ODBC to open the ODBC Data Source Administrator dialog box.

3. Click Add.

4. In the Create New Data Source dialog box, select Microsoft Visual FoxPro Driver and then click Finish.

5. In the **ODBC Visual FoxPro Setup dialog box**, type the data source name and description, select the database type, select the database or directory, and then click OK.

   The new data source name is displayed in the User Data Sources list in the User DSN tab of the ODBC Data Source Administrator dialog box.

6. Click OK to save the new data source and close the ODBC Data Source Administrator dialog box.

© 2016 Microsoft

Modifying a Visual FoxPro Data Source

To modify a Visual FoxPro data source


2. Double-click Data Sources (ODBC) to open the ODBC Data Source Administrator dialog box. This icon is available after you've installed the Visual FoxPro ODBC Driver or any ODBC driver software.

   □ Note
   
   If you are running a previous version of Windows, open the Windows Control Panel and double-click 32-bit ODBC or ODBC to open the ODBC Data Source Administrator dialog box.

3. In the User DSN or System DSN tab, select the name of the data source you want to modify and click Configure.

4. In the **ODBC Visual FoxPro Setup dialog box**, select and change the items you want to modify and then click OK.

5. Click OK to save changes and close the ODBC Data Source Administrator dialog box.

   Your changes will take effect the next time you access the data source from your application.

© 2016 Microsoft

Deleting a Visual FoxPro Data Source

To delete a Visual FoxPro data source


2. Double-click Data Sources (ODBC) to open the ODBC Data Source Administrator dialog box. This icon is available after you've installed the Visual FoxPro ODBC Driver or any ODBC driver software.

   □ Note
   
   If you are running a previous version of Windows, open the Windows Control Panel and double-click 32-bit ODBC or ODBC to open the ODBC Data Source Administrator dialog box.

3. In the User DSN or System DSN tab, select the name of the data source you want to delete.

4. Click Remove.

5. Click OK to save changes and close the ODBC Data Source Administrator dialog box.

© 2016 Microsoft
Connecting to a Visual FoxPro Data Source

You can connect to a Visual FoxPro data source using your Microsoft Office application or using the SQL API.

<table>
<thead>
<tr>
<th>To connect from</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Access, Microsoft Excel, or Microsoft Word</td>
<td>Accessing Visual FoxPro Data from Microsoft Office</td>
</tr>
<tr>
<td>Your C or C++ application</td>
<td>SQLConnect</td>
</tr>
<tr>
<td></td>
<td>SQLDriverConnect</td>
</tr>
<tr>
<td>Your Visual Basic application</td>
<td>Using the Visual FoxPro ODBC Driver with Your Visual Basic Application</td>
</tr>
</tbody>
</table>

Using Connection Strings

You can use a connection string to connect to a Visual FoxPro data source.

For example, to connect to the TasTrade data source and override the current setting of Exclusive associated with the data source, you would use the string:

```
DSN=TasTrade;Exclusive=Yes
```

For a list of the attribute keywords and values you can include in the connection string, see SQLDriverConnect.

For a complete explanation of connection string syntax, see SQLBrowseConnect in the ODBC Programmer's Reference.

Using the VFP FoxPro ODBC Driver with Your Visual Basic Application

Your Microsoft® Visual Basic® application can communicate with Visual FoxPro data by creating a data control that connects to a Visual FoxPro data source.

To connect to Visual FoxPro data using the Data Control in Visual Basic

1. Create a data source named "test" that connects to the TasTrade sample database included in Visual FoxPro. The default Visual FoxPro installation places the TasTrade sample database in the location:

   \c:\vfp\samples\mainsamp\data\tastrade.dbc

2. In Visual Basic, create a new form and place a text box and a Data control on it.

3. Change the Data control's Connect property as follows:

   `ODBC;DATABASE=tastrade;DSN=test`

4. Change the RecordsetType property to the following:

   `2 - Snapshot`

5. Change the RecordSource property to the following:

   `customer`
6. Change the DataSource property for the text box to the default name for the Data control to the following:

   data

7. Change the text box's DataField property to the following:

   customer_id

8. Run the form, and use the Data control to skip through the customer id fields from the Visual FoxPro TasTrade sample database.

© 2016 Microsoft

Using the Visual FoxPro ODBC Driver with Your C or Visual C++ Application

Your C or C++ application communicates with Visual FoxPro data by sending a SQLExecute or SQLExecDirect statement to Visual FoxPro. This statement can contain the following:

- SQL statements native to the Visual FoxPro language, such as the DROP TABLE command.
- Supported ODBC SQL grammar.
- Non-SQL Visual FoxPro language such as supported SET commands.

For more information about SQL native to Visual FoxPro, see the Visual FoxPro documentation.

Example: Using the Visual FoxPro ODBC Driver with Your C or C++ Application

The following example uses the ODBC C API to retrieve data stored in the last_name field in the employee table in the Microsoft® Visual FoxPro sample database named TasTrade. This database is provided with Visual FoxPro and is installed by default in the following location:

c:\vfp\samples\mainsamp\data\tastrade.dbc

The example displays one last name at a time, allowing you to click OK on the message box to see the next last name. It is assumed that a data source named Tastrade has been set up to use the Tastrade.dbc database.

Note

Error checking should be performed on all ODBC API calls; this example excludes error checking for the sake of brevity.

```cpp
// FoxPro_ODBC_Driver_with_C.cpp
// compile with: odbc32.lib user32.lib /c
#include <windows.h>
#include <sql.h>
#include <sqlext.h>
#include <stdio.h>
#include <mbstring.h>
#define MAX_DATA 100
#define MYSQLSUCCESS(rc) ((rc==SQL_SUCCESS)||(rc==SQL_SUCCESS_WITH_INFO))

class direxec {
   RETCODE rc; // ODBC return code
   HENV henv; // Environment
   HDBC hdbc; // Connection handle
   HSTMT hstmt; // Statement handle
   unsigned char szData[MAX_DATA]; // Returned data storage
   SDWORD cbData; // Output length of data
   unsigned char chr_ds_name[SQL_MAX_DSN_LENGTH]; // Data source name

public:
   direxec(); // Constructor
   void sqconn(); // Allocate env, stat, and conn
   void sqexec(unsigned char *); // Execute SQL statement
   void sqdisconn(); // Free pointers to env, stat, conn, and disconnect
   void error_out(); // Displays errors
};
```
// Constructor initializes the string chr_ds_name with the data source name.
direxec::direxec() {
    _mbscpy_s(chr_ds_name, (const unsigned char *)"tastrade");
}

// Allocate environment handle, allocate connection handle,
// connect to data source, and allocate statement handle.
void direxec::sqlconn() {
    SQLAllocEnv(&henv);
    SQLAllocConnect(henv, &hdbc);
    rc=SQLConnect(hdbc, chr_ds_name, SQL_NTS, NULL, 0, NULL, 0);
    // Deallocate handles, display error message, and exit.
    if (!MYSQLSUCCESS(rc)) {
        SQLFreeEnv(henv);
        SQLFreeConnect(hdbc);
        error_out();
        exit(-1);
    }
    rc = SQLAllocStmt(hdbc, &hstmt);
}

// Execute SQL command with SQLExecDirect() ODBC API.
void direxec::sqlexec(unsigned char * cmdstr) {
    rc = SQLExecDirect(hstmt, cmdstr, SQL_NTS);
    if (!MYSQLSUCCESS(rc)) { // Error
        error_out();
        // Deallocate handles and disconnect.
        SQLFreeStmt(hstmt, SQL_DROP);
        SQLDisconnect(hdbc);
        SQLFreeConnect(hdbc);
        SQLFreeEnv(henv);
        exit(-1);
    } else {
        for (rc = SQLFetch(hstmt) ; rc == SQL_SUCCESS ; rc = SQLFetch(hstmt)) {
            SQLGetData(hstmt, 1, SQL_C_CHAR, szData, sizeof(szData), &cbData);
            // In this example, the data is returned in a messagebox for
            // simplicity. However, normally the SQLBindCol() ODBC API could
            // be called to bind individual data rows and assign for a rowset.
            MessageBox(NULL, (const char *)szData, "ODBC", MB_OK);
        }
    }
}

// Free the statement handle, disconnect, free the connection handle, and
// free the environment handle.
void direxec::sqldisconn() {
    SQLFreeStmt(hstmt, SQL_DROP);
    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
}

// Display error message in a messagebox that has an OK button.
void direxec::error_out() {
    unsigned char szSQLSTATE[10];
    SDWORD nErr;
    unsigned char msg[SQL_MAX_MESSAGE_LENGTH + 1];
    SQLWCHAR cbmsg;
    while(SQLError(0, 0, hstmt, szSQLSTATE, &nErr, msg, sizeof(msg), &cbmsg) == SQL_SUCCESS) {
        sprintf_s((char *)szData, MAX_DATA, "Error: SQLSTATE='%s', Native error=%ld, msg='%s'",
                     szSQLSTATE, nErr, msg);
        MessageBox(NULL, (const char *)szData, "ODBC Error", MB_OK);
    }
}

int main() {
    // Declare an instance of the direxec object.
    direxec x;
    // Allocate handles and connect.
    x.sqlconn();
    // Execute SQL command "SELECT last_name FROM employee".
    x.sqlexec((UCHAR FAR *)"SELECT last_name FROM employee");
    // Free handles, and disconnect.
    x.sql disconnect();
    // Return success code; example executed successfully.
    return (TRUE);
}
**Accessing Visual FoxPro Data from Microsoft Office**

You can use the Microsoft Visual FoxPro ODBC Driver to access Visual FoxPro data from your Microsoft Office for Windows 95 or Windows 98 applications.

This section contains the following topics.

<table>
<thead>
<tr>
<th>To</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Microsoft Access</td>
<td>Querying and Updating Visual FoxPro Data from Access</td>
</tr>
<tr>
<td></td>
<td>Importing Visual FoxPro Data into Microsoft Access</td>
</tr>
<tr>
<td>Use Microsoft Excel</td>
<td>Accessing a Visual FoxPro Data Source from Microsoft Excel</td>
</tr>
<tr>
<td></td>
<td>Importing Data into Microsoft Excel from a Visual FoxPro Database</td>
</tr>
<tr>
<td>Use Microsoft Word</td>
<td>Creating Mailing Labels in Microsoft Word Using Visual FoxPro Data</td>
</tr>
</tbody>
</table>

**Querying and Updating Visual FoxPro Data from Microsoft Access**

You can query and update data stored in a Visual FoxPro database from a Microsoft Access database by using the Link Table option.

**To link a Visual FoxPro database to a Microsoft Access database**

1. Open a Microsoft Access database.
2. From the Tables tab, click New.
3. In the New Table dialog box, select Link Table and click OK.
4. In the Link Dialog box, select ODBC Database in the Files of type list.
5. In the SQL Data Sources dialog box, select the data source that connects to the Visual FoxPro data you want to query and click OK.
6. In the Link Tables dialog box, select the tables you want to query and update and click OK. The linked Visual FoxPro tables are displayed in the Tables tab of the Microsoft Access database.

You can now use Microsoft Access to query and update data in the linked Visual FoxPro tables. Changes you make to linked data are sent back to the Visual FoxPro data source.

If you do not want changes you make in Microsoft Access to affect the data on the Visual FoxPro data source, see Importing Visual FoxPro Data into Microsoft Access.

**Importing Visual FoxPro Data into Microsoft Access**

You can import data stored in a Visual FoxPro database into a Microsoft Access database using the Import option.

**To import Visual FoxPro data into a Microsoft Access database**

1. Open a Microsoft Access database.
2. From the File menu, choose Get External Data then Import.
3. In the Import dialog box, select ODBC Databases in the Files of type list.
4. In the SQL Data Sources dialog box, select the Visual FoxPro data source that connects to the FoxPro data you want to query and click OK.
5. In the Import Objects dialog box, select one or more tables you want to import and click OK. The names of the Visual FoxPro tables you imported are
You can now use Microsoft Access to manipulate the data in the imported Visual FoxPro tables. The data you import is a snapshot of the data stored in Visual FoxPro; changes you make to imported data are not sent back to the Visual FoxPro data source.

If you want changes you make in Microsoft Access to change the data on the Visual FoxPro data source, see Querying and Updating Visual FoxPro Data from Microsoft Access.

Accessing a Visual FoxPro Data Source from Microsoft Excel

If you have Microsoft Query installed, you can create a data source in Microsoft Excel that connects to Visual FoxPro data.

To access Visual FoxPro data from Microsoft Excel

1. Open a Microsoft Excel spreadsheet.
2. From the Data menu, choose Get External Data. Microsoft Query opens.
3. In the Select Data Source dialog box, click Other.
4. In the ODBC Data Sources dialog box, click New.
5. In the Add Data Source dialog box, select Microsoft Visual FoxPro Driver from the Installed ODBC Drivers list box and click OK.
6. In the ODBC Visual FoxPro Setup dialog box, enter the data source name, select the Database type, enter the path to the database or directory, and click OK.
7. Click OK.
8. Click Use.

You can now add tables to the open query. For more information about building a query, see Importing Data into Microsoft Excel from a Visual FoxPro Database.

Importing Data into Microsoft Excel from a Visual FoxPro Database

You can import Visual FoxPro data into your Microsoft Excel worksheet if you have defined a data source for it. For information about creating a Visual FoxPro data source, see Accessing a Visual FoxPro Data Source from Microsoft Excel.

To import Visual FoxPro data into an Microsoft Excel worksheet

1. Open a Microsoft Excel spreadsheet.
2. From the Data menu, choose Get External Data. Microsoft Query opens.
3. In the Select Data Source dialog box, select a Visual FoxPro data source and then click Use.
4. If the database accessed by your data source includes tables, select a table from the Add Tables dialog box. Microsoft Query displays the added table in the top half of the query designer.
5. Select fields for your query by dragging them from the table onto the lower half of the designer.
6. Close Microsoft Query. The data you selected is imported into your Microsoft Excel spreadsheet.

Creating Mailing Labels in Microsoft Word Using Visual
**FoxPro Data**

You can use Visual FoxPro data in a Microsoft Word for Windows 95 or Windows 98 document. For example, you might want to create mailing labels from the customer information stored in a Visual FoxPro table.

**To create mailing labels**

1. In Microsoft Word, create a new blank document.
2. From the Tools menu, choose Mail Merge.
3. In the Mail Merge Helper, choose Create and then select Mailing Labels.
5. Under Data Source, choose Get Data and then select Open Data Source.
6. In the Open Data Source dialog box, choose MS Query.
7. In the Select Data Source dialog box, select a Visual FoxPro data source and then click Use.
8. If the database accessed by your data source includes tables, select a table from the Add Tables dialog box. Microsoft Query displays the added table in the top half of the query designer.
9. Select fields for your query by dragging them from the table onto the lower half of the designer.
10. From the File menu, choose Return Data to Microsoft Word. Microsoft Query closes, and the data you selected is available for use in your mail merge document.
12. In the Label Options dialog box, select the printer and label information you want and then click OK.
13. In the Create Labels dialog box, select the fields you want to print on the mailing labels and then click OK.
14. In the Mail Merge Helper, under the Merge the Data with the Document, click Merge.

---

**Visual FoxPro ODBC Driver Programmer's Reference**

This section of the Visual FoxPro ODBC Driver documentation contains the following topics:

- Bookmark Support
- Supported Concurrency Model
- Supported Cursor Model
- Supported Data Types
- International Support
- Support for Rules, Triggers, Default Values, and Stored Procedures
- Supported ODBC SQL Grammar
- Registry Entries
- Supported Scalar Functions
- Supported SET Commands
- Thread Support
- Unsupported FoxPro Commands and Functions
- Error Messages
- Troubleshooting

© 2016 Microsoft
Bookmark Support (Visual FoxPro ODBC Driver)

The Visual FoxPro ODBC Driver supports simple bookmarks. When you call SQLGetInfo with the SQL_BOOKMARK_PERSISTENCE InfoType, the return value is SQL_BP_SCROLL.

For more information about bookmarks, see Bookmarks (ODBC).

Supported Concurrency Model (Visual FoxPro ODBC Driver)

The Visual FoxPro ODBC Driver supports read-only concurrency. Your application can call SQLSetStmtOption with a SQL_CONCURRENCY option of SQL_CONCUR_READONLY.

For more information, see the ODBC Programmer's Reference.

read-only concurrency

The cursor cannot be updated.

row versioning

Essentially timestamp support, in which row versions are compared at update time.

Supported Cursor Model (Visual FoxPro ODBC Driver)

The Visual FoxPro ODBC Driver supports both block (rowset) and static cursors. Static cursors are supported for any driver that conforms to Level 1 ODBC compliance. The driver does not support dynamic, keyset-driven, or mixed (keyset and dynamic) cursors.

Your application can call SQLSetStmtOption with a SQL_CURSOR_TYPE option of SQL_CURSOR_FORWARD_ONLY (block cursor) or SQL_CURSOR_STATIC (static cursor).

If you call SQLSetStmtOption with a SQL_CURSOR_TYPE option other than SQL_CURSOR_FORWARDONLY or SQL_CURSOR_STATIC, the function returns SQL_SUCCEED_WITH_INFO with a SQLSTATE of 01S02 (Option value changed). The driver sets all unsupported cursor modes to SQL_CURSOR_STATIC.

For more information about cursor types and about SQLSetStmtOption, see the ODBC Programmer's Reference.

block cursor

A forward-scrolling, read-only result set returned to the client, who is responsible for maintaining storage for the data.

static cursor

A snapshot of a data set defined by the query. Static cursors do not reflect real-time changes of the underlying data by other users. The cursor’s memory buffer is maintained by the ODBC cursor library, which allows forward and backward scrolling.

rowset

Blocks of data stored in a cursor, representing rows retrieved from a data source.
Supported Data Types (Visual FoxPro ODBC Driver)

The list of data types supported by the driver are presented through the ODBC API and in Microsoft Query.

Data Types in C Applications

You can obtain a list of data types supported by the Visual FoxPro ODBC Driver by using the `SQLGetTypeInfo` function in C or C++ applications.

Data Types in Applications Using Microsoft Query

If your application uses Microsoft Query to create a new table on a Visual FoxPro data source, Microsoft Query displays the New Table Definition dialog box. Under Field Description, the Type box lists Visual FoxPro field data types, represented by single characters.

International Support (Visual FoxPro ODBC Driver)

The Microsoft Visual FoxPro ODBC Driver supports:

- Double-byte character sets (DBCS)
- Multiple collating sequences

A collating sequence defines the sort order for data stored in a Visual FoxPro table or database. By default, the driver is configured to use the collating sequences that support the language version of your operating system.

For a list of supported collating sequences, see `SET COLLATE`.

locale

The set of information that corresponds to a given language and country/region. A locale indicates specific settings such as decimal separators, date and time formats, and character-sorting order.

sort order

Sort orders incorporate the sorting rules of different locales, allowing you to sort data in those languages correctly. In Visual FoxPro, the current sort order determines the results of character expression comparisons and the order in which the records appear in indexed or sorted tables.

Support for Rules, Triggers, Default Values, and Stored Procedures (Visual FoxPro ODBC Driver)

You cannot create Visual FoxPro rules, triggers, default values, or stored procedures using the Visual FoxPro ODBC Driver. However, your application might interact with existing rules, triggers, default values, or stored procedures as it inserts, updates, or deletes Visual FoxPro data stored in a database.

The following table lists the Visual FoxPro commands and functions supported by the Visual FoxPro ODBC Driver when the commands or functions exist in rules, triggers, default values, or stored procedures.

<table>
<thead>
<tr>
<th>Command/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VERSION()</code></td>
<td>Returns &quot;Visual FoxPro ODBC Driver &lt;version&gt;&quot; when called by the driver.</td>
</tr>
</tbody>
</table>

Tip

If you want to insert conditional code into your rules, triggers, or stored procedures that determines the commands to execute when called by the driver, you can use the `VERSION()` function.
### Visual FoxPro Commands and Functions Supported in Rules, Triggers, Default Values, and Stored Procedures

<table>
<thead>
<tr>
<th>$ Operator</th>
<th>% Operator</th>
<th>&amp; Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>B &amp; Command</td>
<td>* Command</td>
<td>= Command</td>
</tr>
</tbody>
</table>

#### A

<table>
<thead>
<tr>
<th>ABS( ) Function</th>
<th>ACOPY( ) Function</th>
<th>ADD TABLE Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADATABASES( ) Function</td>
<td>ADBOBJECTS( ) Function</td>
<td>AERROR( ) Function</td>
</tr>
<tr>
<td>ADEL( ) Function</td>
<td>AELEMENT( ) Function</td>
<td>ALEN( ) Function</td>
</tr>
<tr>
<td>AFIELDS( ) Function</td>
<td>AINS( ) Function</td>
<td>ALTER TABLE - SQL Command</td>
</tr>
<tr>
<td>ALIAS( ) Function</td>
<td>ALLTRIM( ) Function</td>
<td>APPEND FROM ARRAY Command</td>
</tr>
<tr>
<td>AND Operator</td>
<td>APPEND Command</td>
<td>APPEND MEMO Command</td>
</tr>
<tr>
<td>APPEND FROM Command</td>
<td>APPEND GENERAL Command</td>
<td>ASCAN( ) Function</td>
</tr>
<tr>
<td>APPEND PROCEDURES Command</td>
<td>ASC( ) Function</td>
<td>ASUBSCRIPT( ) Function</td>
</tr>
<tr>
<td>ASIN( ) Function</td>
<td>ASORT( ) Function</td>
<td>ATAN( ) Function</td>
</tr>
<tr>
<td>AT( ) Function</td>
<td>AT_C( ) Function</td>
<td>ATCLINE( ) Function</td>
</tr>
<tr>
<td>ATC( ) Function</td>
<td>ATCC( ) Function</td>
<td>AUSED( ) Function</td>
</tr>
<tr>
<td>ATLINE( ) Function</td>
<td>ATN2( ) Function</td>
<td></td>
</tr>
<tr>
<td>AVERAGE Command</td>
<td>ACOS( ) Function</td>
<td></td>
</tr>
</tbody>
</table>

#### B

<table>
<thead>
<tr>
<th>BEGIN TRANSACTION Command</th>
<th>BETWEEN( ) Function</th>
<th>BITNOT( ) Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITCLEAR( ) Function</td>
<td>BITLSHIFT( ) Function</td>
<td>BITSET( ) Function</td>
</tr>
<tr>
<td>BITOR( ) Function</td>
<td>BITRSHIFT( ) Function</td>
<td>BLANK Command</td>
</tr>
<tr>
<td>BITTEST( ) Function</td>
<td>BITXOR( ) Function</td>
<td></td>
</tr>
<tr>
<td>BOF( ) Function</td>
<td>BITAND( ) Function</td>
<td></td>
</tr>
</tbody>
</table>

#### C

<table>
<thead>
<tr>
<th>CALCULATE Command</th>
<th>CANDIDATE( ) Function</th>
<th>CHR( ) Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDX( ) Function</td>
<td>CEILING( ) Function</td>
<td>CLOSE Commands</td>
</tr>
<tr>
<td>CHRTRAN( ) Function</td>
<td>CHRTRANC( ) Function</td>
<td>COPY INDEXES Command</td>
</tr>
<tr>
<td>CMONTH( ) Function</td>
<td>CONTINUE Command</td>
<td>COPY STRUCTURE EXTENDED Command</td>
</tr>
<tr>
<td>COPY PROCEDURES Command</td>
<td>COPY STRUCTURE Command</td>
<td>COPY TO Command</td>
</tr>
<tr>
<td>COPY TAG Command</td>
<td>COPY TO ARRAY Command</td>
<td>CPCONVERT( ) Function</td>
</tr>
<tr>
<td>COS( ) Function</td>
<td>COUNT Command</td>
<td>CTOD( ) Function</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>IDXCOLLATE( ) Function</td>
<td>IF ... ENDIF Command</td>
<td>IIF( ) Function</td>
</tr>
<tr>
<td>INDBC( ) Function</td>
<td>INDEX Command</td>
<td>INLIST( ) Function</td>
</tr>
<tr>
<td>INSERT-SQL Command</td>
<td>INT( ) Function</td>
<td>ISALPHA( ) Function</td>
</tr>
<tr>
<td>ISBLANK( ) Function</td>
<td>ISDIGIT( ) Function</td>
<td>ISEXCLUSIVE( ) Function</td>
</tr>
<tr>
<td>ISLEADBYTE( ) Function</td>
<td>ISLOWER( ) Function</td>
<td>ISNULL( ) Function</td>
</tr>
<tr>
<td>ISREADONLY( ) Function</td>
<td>ISUPPER( ) Function</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY( ) Function</td>
<td>KEYMATCH( ) Function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFT( ) Function</td>
<td>LEFTC( ) Function</td>
</tr>
<tr>
<td>LENC( ) Function</td>
<td>LIKE( ) Function</td>
</tr>
<tr>
<td>LOCAL Command</td>
<td>LOCATE Command</td>
</tr>
<tr>
<td>LOG( ) Function</td>
<td>LOG10( ) Function</td>
</tr>
<tr>
<td>LOWER( ) Function</td>
<td>LPARAMETERS Command</td>
</tr>
<tr>
<td>LUPDATE( ) Function</td>
<td>LEN( ) Function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_MLINE System Memory Variable</td>
<td>MAX( ) Function</td>
</tr>
<tr>
<td>MDY( ) Function</td>
<td>MEMLINES( ) Function</td>
</tr>
<tr>
<td>MIN( ) Function</td>
<td>MINUTE( ) Function</td>
</tr>
<tr>
<td>MOD( ) Function</td>
<td>MONTH( ) Function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NDX( ) Function</td>
<td>NORMALIZE( ) Function</td>
</tr>
<tr>
<td>NOTE Command</td>
<td>NTOM( ) Function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OCCURS( ) Function</td>
<td>OLDVAL( ) Function</td>
</tr>
<tr>
<td>ON KEY Command</td>
<td>ON( ) Function</td>
</tr>
<tr>
<td>OR Operator</td>
<td>ORDER( ) Function</td>
</tr>
</tbody>
</table>
### P

<table>
<thead>
<tr>
<th>Command/Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACK Command</td>
</tr>
<tr>
<td>PARAMETERS( ) Function</td>
</tr>
<tr>
<td>PAYMENT( ) Function</td>
</tr>
<tr>
<td>PARAMETERS Command</td>
</tr>
<tr>
<td>PRIMARY( ) Function</td>
</tr>
<tr>
<td>PRIVATE Command</td>
</tr>
<tr>
<td>PI( ) Function</td>
</tr>
<tr>
<td>PROGRAM( ) Function</td>
</tr>
<tr>
<td>PROPER( ) Function</td>
</tr>
<tr>
<td>PROCEDURE Command</td>
</tr>
<tr>
<td>PV( ) Function</td>
</tr>
<tr>
<td>PUBLIC Command</td>
</tr>
<tr>
<td>PADL( )</td>
</tr>
</tbody>
</table>

### R

<table>
<thead>
<tr>
<th>Command/Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAND( ) Function</td>
</tr>
<tr>
<td>RAT( ) Function</td>
</tr>
<tr>
<td>RATC( ) Function</td>
</tr>
<tr>
<td>RATLINE( ) Function</td>
</tr>
<tr>
<td>RECALL Command</td>
</tr>
<tr>
<td>RECCOUNT( ) Function</td>
</tr>
<tr>
<td>RECNO( ) Function</td>
</tr>
<tr>
<td>RECSIZE( ) Function</td>
</tr>
<tr>
<td>REGIONAL Command</td>
</tr>
<tr>
<td>RELATION( ) Function</td>
</tr>
<tr>
<td>REPLACE FROM ARRAY Command</td>
</tr>
<tr>
<td>REPLICA( ) Function</td>
</tr>
<tr>
<td>RETRY Command</td>
</tr>
<tr>
<td>RETURN Command</td>
</tr>
<tr>
<td>RIGHT( ) Function</td>
</tr>
<tr>
<td>RIGHTC( ) Function</td>
</tr>
<tr>
<td>RLOCK( ) Function</td>
</tr>
<tr>
<td>ROLLBACK Command</td>
</tr>
<tr>
<td>ROUND( ) Function</td>
</tr>
<tr>
<td>RTOD( ) Function</td>
</tr>
<tr>
<td>RTRIM( ) Function</td>
</tr>
</tbody>
</table>

### S

<table>
<thead>
<tr>
<th>Command/Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAN ... ENDSCAN Command</td>
</tr>
<tr>
<td>SCATTER Command</td>
</tr>
<tr>
<td>SEC( ) Function</td>
</tr>
<tr>
<td>SECONDS( ) Function</td>
</tr>
<tr>
<td>SEEK Command</td>
</tr>
<tr>
<td>SEEK( ) Function</td>
</tr>
<tr>
<td>SELECT Command</td>
</tr>
<tr>
<td>SELECTC( ) Function</td>
</tr>
<tr>
<td>SELECT-SQL Command</td>
</tr>
<tr>
<td>SET BLOCKSIZE Command</td>
</tr>
<tr>
<td>SET CARRY Command</td>
</tr>
<tr>
<td>SET CENTURY Command</td>
</tr>
<tr>
<td>SET COLLATE Command</td>
</tr>
<tr>
<td>SET DATABASE Command</td>
</tr>
<tr>
<td>SET DATE Command</td>
</tr>
<tr>
<td>SET DEFAULT Command</td>
</tr>
<tr>
<td>SET DELETED Command</td>
</tr>
<tr>
<td>SET EXACT Command</td>
</tr>
<tr>
<td>SET EXCLUSIVE Command</td>
</tr>
<tr>
<td>SET FDOW Command</td>
</tr>
<tr>
<td>SET FIELDS Command</td>
</tr>
<tr>
<td>SET FILTER Command</td>
</tr>
<tr>
<td>SET FIXED Command</td>
</tr>
<tr>
<td>SET FULLPATH Command</td>
</tr>
<tr>
<td>SET FWEEK Command</td>
</tr>
<tr>
<td>SET HOURS Command</td>
</tr>
<tr>
<td>SET INDEX Command</td>
</tr>
<tr>
<td>SET LOCK Command</td>
</tr>
<tr>
<td>SET MULTILOCKS Command</td>
</tr>
<tr>
<td>SET NEAR Command</td>
</tr>
<tr>
<td>SET NOCPTRANS Command</td>
</tr>
<tr>
<td>SET NOTIFY Command</td>
</tr>
<tr>
<td>SET NULL Command</td>
</tr>
<tr>
<td>SET OPTIMIZE Command</td>
</tr>
<tr>
<td>SET ORDER Command</td>
</tr>
<tr>
<td>SET PATH Command</td>
</tr>
<tr>
<td>SET PROCEDURE Command</td>
</tr>
<tr>
<td>SET RELATION Command</td>
</tr>
<tr>
<td>SET RELATION OFF Command</td>
</tr>
<tr>
<td>SET REPROCESS Command</td>
</tr>
<tr>
<td>SET SKIP Command</td>
</tr>
<tr>
<td>SET UDFPARMS Command</td>
</tr>
<tr>
<td>SET UNIQUE Command</td>
</tr>
<tr>
<td>SET VOLUME Command</td>
</tr>
<tr>
<td>SET( ) Function</td>
</tr>
<tr>
<td>SETFLDSTATE( ) Function</td>
</tr>
<tr>
<td>SIGN( ) Function</td>
</tr>
<tr>
<td>SIN( ) Function</td>
</tr>
<tr>
<td>SKIP Command</td>
</tr>
<tr>
<td>SORT Command</td>
</tr>
<tr>
<td>SPACE( ) Function</td>
</tr>
<tr>
<td>SQRT( ) Function</td>
</tr>
<tr>
<td>STORE Command</td>
</tr>
<tr>
<td>STR( ) Function</td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
<tr>
<td>STRCONV( )</td>
</tr>
<tr>
<td>STUFFC( )</td>
</tr>
<tr>
<td>SUM Command</td>
</tr>
</tbody>
</table>

**T**

<table>
<thead>
<tr>
<th>System Memory Variable</th>
<th>System Memory Variable</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>_TALLY</td>
<td>_TRIGGERLEVEL</td>
<td>TAGCOUNT( )</td>
</tr>
<tr>
<td>TABLEUPDATE( ) Function</td>
<td>TAG( ) Function</td>
<td>TARGET( ) Function</td>
</tr>
<tr>
<td>TAGNO( ) Function</td>
<td>TAN( ) Function</td>
<td>TRIM( ) Function</td>
</tr>
<tr>
<td>TIME( ) Function</td>
<td>TOTAL Command</td>
<td>TXNLEVEL( ) Function</td>
</tr>
<tr>
<td>TTOC( ) Function</td>
<td>TTOD( ) Function</td>
<td></td>
</tr>
<tr>
<td>TYPE( ) Function</td>
<td>TABLEREVERT( ) Function</td>
<td></td>
</tr>
</tbody>
</table>

**U**

<table>
<thead>
<tr>
<th>Function</th>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIQUE( ) Function</td>
<td>UNLOCK Command</td>
<td>USE Command</td>
</tr>
<tr>
<td>UPDATE Command</td>
<td>UPPER( ) Function</td>
<td></td>
</tr>
<tr>
<td>USED( ) Function</td>
<td>UPDATE - SQL Command</td>
<td></td>
</tr>
</tbody>
</table>

**V**

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAL( ) Function</td>
<td>VERSION( ) Function</td>
</tr>
</tbody>
</table>

**W**

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEEK( ) Function</td>
</tr>
</tbody>
</table>

**Y**

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR( ) Function</td>
</tr>
</tbody>
</table>

**Z**

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZAP Command</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**Supported ODBC SQL Grammar (Visual FoxPro ODBC Driver)**

The Microsoft Visual FoxPro ODBC Driver supports the following:
• All SQL statements and clauses in the ODBC minimum SQL grammar
• An additional SQL statement from the ODBC core SQL grammar

The following table lists the items supported by the driver, by ODBC SQL Grammar level.

<table>
<thead>
<tr>
<th>Level</th>
<th>Elements</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>Data Definition Language (DDL)</td>
<td>CREATE TABLE and DROP TABLE</td>
</tr>
<tr>
<td></td>
<td>Data Manipulation Language (DML)</td>
<td>SELECT, INSERT, UPDATE, and DELETE</td>
</tr>
<tr>
<td></td>
<td>Expressions</td>
<td>Simple (such as A&gt;B+C)</td>
</tr>
<tr>
<td></td>
<td>Data types</td>
<td>CHAR, VARCHAR, or LONG VARCHAR</td>
</tr>
</tbody>
</table>

In addition to the supported ODBC SQL grammar, the Visual FoxPro ODBC Driver supports the complete native Visual FoxPro language syntax for the following Visual FoxPro commands:

ALTER TABLE
CREATE TABLE
DELETE
DELETE TAG
DROP TABLE
INDEX
INSERT
SELECT
UPDATE

© 2016 Microsoft

Registry Entries (Visual FoxPro ODBC Driver)

When you install the Visual FoxPro ODBC Driver, the installation program updates your system's registry, in the registry key HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCInst.ini, to add a new key called Microsoft Visual FoxPro Driver. Under that key, the values described in the following table are added.

<table>
<thead>
<tr>
<th>Value name</th>
<th>Value type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>APILevel</td>
<td>REG_SZ</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>ConnectFunctions</td>
<td>REG_SZ</td>
<td>&quot;YYN&quot;</td>
</tr>
<tr>
<td>Driver</td>
<td>REG_SZ</td>
<td>System path to the vfpodbc.dll file</td>
</tr>
<tr>
<td>DriverODBCVer</td>
<td>REG_SZ</td>
<td>&quot;02.50&quot;</td>
</tr>
<tr>
<td>FileExtns</td>
<td>REG_SZ</td>
<td>&quot;<em>.dbf,</em>.cdx,*.fpt&quot;</td>
</tr>
<tr>
<td>FileUsage</td>
<td>REG_SZ</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>Setup</td>
<td>REG_SZ</td>
<td>System path to the vfpodbc.dll file</td>
</tr>
<tr>
<td>SQLLevel</td>
<td>REG_SZ</td>
<td>&quot;0&quot;</td>
</tr>
</tbody>
</table>

The installation program also adds the key "Visual FoxPro Files", representing the default Visual FoxPro driver, to your system's HKEY_CURRENT_USER\SOFTWARE\ODBC\Odbc.ini key. Under this key, the installation program adds the values described in the following table.

<table>
<thead>
<tr>
<th>Value name</th>
<th>Value type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>REG_SZ</td>
<td>System path to the vfpodbc.dll file</td>
</tr>
</tbody>
</table>
Each time you add a Visual FoxPro ODBC data source to your ODBC configuration, a new key is added for that data source name. The values for the data source correspond to values you set in the ODBC Visual FoxPro Setup dialog box, as listed in the following table.

<table>
<thead>
<tr>
<th>Value name (keyword)</th>
<th>Value type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collate</td>
<td>REG_SQ</td>
<td>Any supported collating sequence</td>
</tr>
<tr>
<td>Description</td>
<td>REG_SZ</td>
<td>User description of data source</td>
</tr>
<tr>
<td>Driver</td>
<td></td>
<td>System path to the vfpodbc.dll file</td>
</tr>
<tr>
<td>Exclusive</td>
<td></td>
<td>Yes or No</td>
</tr>
<tr>
<td>BackgroundFetch</td>
<td>REG_SZ</td>
<td>Path to .DBC file</td>
</tr>
<tr>
<td>SourceDB</td>
<td>REG_SZ</td>
<td>&quot;DBC&quot; or &quot;DBF&quot;</td>
</tr>
</tbody>
</table>

You should not access this information directly; any administration of the registry is handled by the ODBC Administrator when you add, modify, or delete a data source.

You can use some of these keywords and values as parameters in the SQLDriverConnect ODBC API function.

Supported Scalar Functions (Visual FoxPro ODBC Driver)

The Visual FoxPro ODBC Driver supports three types of scalar functions as defined in the ODBC SQL grammar: string, numeric, and time and date functions.

This section contains the following topics.

- String Functions (Visual FoxPro ODBC Driver)
- Numeric Functions (Visual FoxPro ODBC Driver)
- Time and Date Functions (Visual FoxPro ODBC Driver)

String Functions (Visual FoxPro ODBC Driver)

The following table lists ODBC string manipulation functions supported by the Visual FoxPro ODBC Driver; when the Visual FoxPro grammar for the same function differs from the ODBC syntax, the Visual FoxPro equivalent is listed.

<table>
<thead>
<tr>
<th>ODBC grammar</th>
<th>Visual FoxPro grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII (string_exp)</td>
<td>ASC (string_exp)</td>
</tr>
<tr>
<td>CHAR (code)</td>
<td>CHR (string_exp)</td>
</tr>
<tr>
<td>CONCAT (string_exp1, string_exp2)</td>
<td>string_exp1 + string_exp2</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>DIFFERENCE</td>
<td>(string_exp1, string_exp2)</td>
</tr>
<tr>
<td>INSERT</td>
<td>(string_exp1, start, length, string_exp2)</td>
</tr>
<tr>
<td>STUFF</td>
<td>(string_exp1, start, length, string_exp2)</td>
</tr>
<tr>
<td>LCASE</td>
<td>(string_exp)</td>
</tr>
<tr>
<td>LOWER</td>
<td>(string_exp)</td>
</tr>
<tr>
<td>LEFT</td>
<td>(string_exp, count)</td>
</tr>
<tr>
<td>LENGTH</td>
<td>(string_exp)</td>
</tr>
<tr>
<td>LEN</td>
<td>(string_exp)</td>
</tr>
<tr>
<td>LTRIM</td>
<td>(string_exp)</td>
</tr>
<tr>
<td>REPEAT</td>
<td>(string_exp, count)</td>
</tr>
<tr>
<td>REPLICATE</td>
<td>(string_exp, count)</td>
</tr>
<tr>
<td>REPLACE</td>
<td>(string_exp1, string_exp2, string_exp3)</td>
</tr>
<tr>
<td>STRTRAN</td>
<td>(string_exp1, string_exp2, string_exp3)</td>
</tr>
<tr>
<td>RIGHT</td>
<td>(string_exp, count)</td>
</tr>
<tr>
<td>RTRIM</td>
<td>(string_exp)</td>
</tr>
<tr>
<td>SOUNDEX</td>
<td>(string_exp)</td>
</tr>
<tr>
<td>SPACE</td>
<td>(count)</td>
</tr>
<tr>
<td>SUBSTRING</td>
<td>(string_exp, start, length)</td>
</tr>
<tr>
<td>SUBSTR</td>
<td>(string_exp, start, length)</td>
</tr>
<tr>
<td>UCASE</td>
<td>(string_exp)</td>
</tr>
<tr>
<td>UPPER</td>
<td>(string_exp)</td>
</tr>
</tbody>
</table>
The following table describes ODBC numeric functions supported by the Visual FoxPro ODBC Driver; when the Visual FoxPro grammar for the same function differs from the ODBC syntax, the Visual FoxPro equivalent is listed.

<table>
<thead>
<tr>
<th>ODBC grammar</th>
<th>Visual FoxPro grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS (numeric_exp)</td>
<td></td>
</tr>
<tr>
<td>ACOS (float_exp)</td>
<td></td>
</tr>
<tr>
<td>ASIN (float_exp)</td>
<td></td>
</tr>
<tr>
<td>ATAN (float_exp)</td>
<td></td>
</tr>
<tr>
<td>ATAN2 (float_exp1, float_exp2)</td>
<td>ATN2 (float_exp1, float_exp2)</td>
</tr>
<tr>
<td>CEILING (numeric_exp)</td>
<td></td>
</tr>
<tr>
<td>COS (float_exp)</td>
<td></td>
</tr>
<tr>
<td>COT (float_exp)</td>
<td></td>
</tr>
<tr>
<td>DEGREES (numeric_exp)</td>
<td>RTOD (numeric_exp)</td>
</tr>
<tr>
<td>EXP (float_exp)</td>
<td></td>
</tr>
<tr>
<td>FLOOR (numeric_exp)</td>
<td></td>
</tr>
<tr>
<td>LOG (float_exp)</td>
<td></td>
</tr>
</tbody>
</table>
The following numeric functions are not supported:

- `POWER (numeric_exp, integer_exp)`
- `TRUNCATE (numeric_exp, integer_exp)`

### Time and Date Functions (Visual FoxPro ODBC Driver)

The following table lists ODBC time and date functions supported by the Visual FoxPro ODBC Driver; when the Visual FoxPro grammar for the same function differs from the ODBC syntax, the Visual FoxPro equivalent is listed.

<table>
<thead>
<tr>
<th>ODBC grammar</th>
<th>Visual FoxPro grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG10 (float_exp)</td>
<td></td>
</tr>
<tr>
<td>MOD (integer_exp1, integer_exp2)</td>
<td></td>
</tr>
<tr>
<td>PI ()</td>
<td></td>
</tr>
<tr>
<td>RADIANS (numeric_exp)</td>
<td>DTOR (numeric_exp)</td>
</tr>
<tr>
<td>RAND ([integer_exp])</td>
<td></td>
</tr>
<tr>
<td>ROUND (numeric_exp, integer_exp)</td>
<td></td>
</tr>
<tr>
<td>SIGN (numeric_exp)</td>
<td></td>
</tr>
<tr>
<td>SIN (float_exp)</td>
<td></td>
</tr>
<tr>
<td>SQRT (float_exp)</td>
<td></td>
</tr>
<tr>
<td>TAN (float_exp)</td>
<td></td>
</tr>
</tbody>
</table>
The following time and date functions are not supported:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAYOFYEAR</td>
<td>(date_exp)</td>
</tr>
<tr>
<td>QUARTER</td>
<td>(date_exp)</td>
</tr>
<tr>
<td>TIMESTAMPADD</td>
<td>(interval, integer_exp, timestamp_exp)</td>
</tr>
<tr>
<td>TIMESTAMPDIFF</td>
<td>(interval, timestamp_exp1, timestamp_exp2)</td>
</tr>
<tr>
<td>CURDATE</td>
<td>()</td>
</tr>
<tr>
<td>DATE</td>
<td>()</td>
</tr>
<tr>
<td>CURTIME</td>
<td>()</td>
</tr>
<tr>
<td>TIME</td>
<td>()</td>
</tr>
<tr>
<td>DAYNAME</td>
<td>(date_exp)</td>
</tr>
<tr>
<td>CDOW</td>
<td>(date_exp)</td>
</tr>
<tr>
<td>DAYOFMONTH</td>
<td>(date_exp)</td>
</tr>
<tr>
<td>DAY</td>
<td>()</td>
</tr>
<tr>
<td>HOUR</td>
<td>(time_exp)</td>
</tr>
<tr>
<td>MINUTE</td>
<td>(time_exp)</td>
</tr>
<tr>
<td>MONTH</td>
<td>(time_exp)</td>
</tr>
<tr>
<td>MONTHNAME</td>
<td>(date_exp)</td>
</tr>
<tr>
<td>CMONTH</td>
<td>(date_exp)</td>
</tr>
<tr>
<td>NOW</td>
<td>()</td>
</tr>
<tr>
<td>DATETIME</td>
<td>()</td>
</tr>
<tr>
<td>SECOND</td>
<td>(time_exp)</td>
</tr>
<tr>
<td>SEC</td>
<td>(time_exp)</td>
</tr>
<tr>
<td>WEEK</td>
<td>(date_exp)</td>
</tr>
<tr>
<td>YEAR</td>
<td>(date_exp)</td>
</tr>
</tbody>
</table>
ODBC Escape Sequences

The driver also supports the ODBC escape sequence for date and timestamp data. The escape clause syntax is as follows:

```
{d 'value'}
{ts 'value'}
```

In this syntax, `d` indicates that value is a date in the `yyyy-mm-dd` format and `ts` indicates that value is a timestamp in the `yyyy-mm-dd hh:mm:ss.[f...]` format. The shorthand syntax for date and timestamp data is as follows:

```
{d 'value'}
{ts 'value'}
```

For example, each of the following statements updates the ALLTYPES table by using the date and timestamp shorthand syntax in a supported SQL UPDATE command:

```
UPDATE alltypes
  SET DAT_COL={d'1968-04-28'}
  WHERE KEY=111
UPDATE alltypes
  SET DTI_COL={ts'1968-04-28 12:00:00'}
  WHERE KEY=111
```

Remarks

For more information about escape sequences, see Escape Sequences in ODBC in the ODBC Programmer's Reference.

© 2016 Microsoft

Supported SET Commands (Visual FoxPro ODBC Driver)

Your application can send the following Visual FoxPro SET commands to a data source:

```
SET ANSI
SET BLOCKSIZE
SET COLLATE
SET DELETED
SET EXACT
SET EXCLUSIVE
SET NULL
SET PATH
SET REPROCESS
SET UNIQUE
```

© 2016 Microsoft

Thread Support (Visual FoxPro ODBC Driver)

The Visual FoxPro ODBC Driver is thread-safe. Access to environment handles (hen), connection handles (hdbc), and statement handles (hstmt) is wrapped in appropriate semaphores to prevent other processes from accessing and potentially altering the driver’s internal data structures.

In a multithreaded application, you can cancel a function that is running synchronously on an hstmt by calling SQLCancel on a separate thread.

The driver uses a separate thread to fetch data when you use progressive fetching. To use progressive fetching for a data source, select the Fetch data in background check box on the ODBC Visual FoxPro Setup dialog box or use the BackgroundFetch attribute keyword in your connection string. Avoid using background fetch when you call the driver from multithreaded applications. For information about connection string attribute keywords, see Using Connection...
For more information about threads and **SQLCancel**, see **SQLCancel** in the ODBC Programmer's Reference.

© 2016 Microsoft

Unsupported Visual FoxPro Commands and Functions (Visual FoxPro ODBC Driver)

The following table lists FoxPro commands and functions that are not supported by the Visual FoxPro ODBC Driver but are supported by Microsoft® Visual FoxPro.

If your application interacts with data whose rules, triggers, default values, or stored procedures call these Visual FoxPro commands or functions, the driver can generate an error.

### Unsupported Visual FoxPro Commands and Functions

<table>
<thead>
<tr>
<th>#DEFINE ... #UNDEF</th>
<th>#IF ... #ENDIF Preprocessor Directive</th>
<th>#IFDEF</th>
<th>#IFNDEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>#INCLUDE Preprocessor Directive</td>
<td>:: Scope Resolution Operator</td>
<td>! Command (see RUN</td>
<td>! Command)</td>
</tr>
<tr>
<td>?</td>
<td>?? Command</td>
<td>??? Command</td>
<td>\ \ \ Command</td>
</tr>
<tr>
<td>@ ... BOX Command</td>
<td>@ ... CLASS Command</td>
<td>@ ... CLEAR Command</td>
<td></td>
</tr>
<tr>
<td>@ ... EDIT - Edit Boxes Command</td>
<td>@ ... FILL Command</td>
<td>@ ... GET</td>
<td></td>
</tr>
<tr>
<td>@ ... MENU Command</td>
<td>@ ... PROMPT Command</td>
<td>@ ... SAY Command</td>
<td></td>
</tr>
<tr>
<td>@ ... SCROLL Command</td>
<td>@ ... TO Command</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A**

<table>
<thead>
<tr>
<th>ACCEPT Command</th>
<th>ACLASS( ) Function</th>
<th>ACTIVATE MENU Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVATE POPUP Command</td>
<td>ACTIVATE SCREEN Command</td>
<td>ACTIVATE WINDOW Command</td>
</tr>
<tr>
<td>ActivateCell Method</td>
<td>ADD CLASS Command</td>
<td>ADIR( ) Function</td>
</tr>
<tr>
<td>AFONT( ) Function</td>
<td>AINSTANCE( ) Function</td>
<td>_ALIGNMENT System Memory Variable</td>
</tr>
<tr>
<td>AMEMBERS( ) Function</td>
<td>ANSITOOEM( ) Function</td>
<td>APRINTERs( ) Function</td>
</tr>
<tr>
<td>ASELOBJ( ) Function</td>
<td>ASSIST Command</td>
<td></td>
</tr>
</tbody>
</table>

**B**

<table>
<thead>
<tr>
<th>BAR( ) Function</th>
<th>BARCOUNT( ) Function</th>
<th>BARPROMPT( ) Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>_BEAUTIFY System Memory Variable</td>
<td>_BOX System Memory Variable</td>
<td>BROWSE Command</td>
</tr>
<tr>
<td>_BROWSER System Memory Variable</td>
<td>BUILD APP Command</td>
<td>BUILD EXE Command</td>
</tr>
<tr>
<td>BUILD PROJECT Command</td>
<td>_BUILDER System Memory Variable</td>
<td></td>
</tr>
</tbody>
</table>

**C**

<table>
<thead>
<tr>
<th>_CALCVALUE System Memory Variable</th>
<th>_CLIPTEXT System Memory Variable</th>
<th>_CONVERTER System Memory Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>_CUROBJ System Memory Variable</td>
<td>CALL Command</td>
<td>CANCEL Command</td>
</tr>
<tr>
<td>CAPSLOCK( ) Function</td>
<td>CD Command</td>
<td>CHANGE Command</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------</td>
<td>----------------</td>
</tr>
<tr>
<td>CHDIR Command</td>
<td>CHRSAW( ) Function</td>
<td>CLOSE MEMO Command</td>
</tr>
<tr>
<td>CNTBAR( ) Function</td>
<td>CNTPAD( ) Function</td>
<td>COL( ) Function</td>
</tr>
<tr>
<td>COMPILE Command</td>
<td>COMPILE DATABASE Command</td>
<td>COMPILE FORM Command</td>
</tr>
<tr>
<td>COMPOBJ( ) Function</td>
<td>Container Object</td>
<td>Control Object</td>
</tr>
<tr>
<td>COPY FILE Command</td>
<td>COPY MEMO Command</td>
<td>CREATE CLASS Command</td>
</tr>
<tr>
<td>CREATE CLASSLIB Command</td>
<td>CREATE COLOR SET Command</td>
<td>CREATE Command</td>
</tr>
<tr>
<td>CREATE CONNECTION Command</td>
<td>CREATE DATABASE Command</td>
<td>CREATE FORM Command</td>
</tr>
<tr>
<td>CREATE FROM Command</td>
<td>CREATE LABEL Command</td>
<td>CREATE MENU Command</td>
</tr>
<tr>
<td>CREATE PROJECT Command</td>
<td>CREATE QUERY Command</td>
<td>CREATE REPORT Command</td>
</tr>
<tr>
<td>CREATE SCREEN Command</td>
<td>CREATE SQL VIEW Command</td>
<td>CREATE TRIGGER Command</td>
</tr>
<tr>
<td>CREATE VIEW Command</td>
<td>CREATEOBJECT( ) Function</td>
<td>CURDIR( ) Function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DBLCCLICK System Memory Variable</td>
</tr>
<tr>
<td>DDE Functions</td>
</tr>
<tr>
<td>DEACTIVATE WINDOW Command</td>
</tr>
<tr>
<td>DEFINE BAR Command</td>
</tr>
<tr>
<td>DEFINE MENU Command</td>
</tr>
<tr>
<td>DEFINE WINDOW Command</td>
</tr>
<tr>
<td>DELETE FILE Command</td>
</tr>
<tr>
<td>DIR Command</td>
</tr>
<tr>
<td>DISPLAY CONNECTIONS Command</td>
</tr>
<tr>
<td>DISPLAY FILES Command</td>
</tr>
<tr>
<td>DISPLAY PROCEDURES Command</td>
</tr>
<tr>
<td>DISPLAY TABLES Command</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDIT Command</td>
</tr>
<tr>
<td>ERASE Command</td>
</tr>
<tr>
<td>EJECT Command</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>_FOXDOC System Memory Variable</td>
</tr>
<tr>
<td>FCLOSE( ) Function</td>
</tr>
<tr>
<td>FERROR( ) Function</td>
</tr>
<tr>
<td>Command</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>FILER Command</td>
</tr>
<tr>
<td>FMAX( ) Function</td>
</tr>
<tr>
<td>FTPUTS( ) Function</td>
</tr>
<tr>
<td>FWRITE( ) Function</td>
</tr>
<tr>
<td>_GENGRAPH System Memory Variable</td>
</tr>
<tr>
<td>_GENSCRN System Memory Variable</td>
</tr>
<tr>
<td>GETCOLOR( ) Function</td>
</tr>
<tr>
<td>GETFILE( ) Function</td>
</tr>
<tr>
<td>GETPAD( ) Function</td>
</tr>
<tr>
<td>HELP Command</td>
</tr>
<tr>
<td>HIDE WINDOW Command</td>
</tr>
<tr>
<td>IMESTATUS( ) Function</td>
</tr>
<tr>
<td>INDEX ON Command</td>
</tr>
<tr>
<td>INSERT Command</td>
</tr>
<tr>
<td>ISMOUSE( ) Function</td>
</tr>
<tr>
<td>JOIN Command</td>
</tr>
<tr>
<td>KEYBOARD Command</td>
</tr>
<tr>
<td>_LMARGIN System Memory Variable</td>
</tr>
<tr>
<td>LINENO( ) Function</td>
</tr>
<tr>
<td>LOAD Command</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>----------------------------------------</td>
</tr>
<tr>
<td>MCOL( ) Function</td>
</tr>
<tr>
<td>MEMORY( ) Function</td>
</tr>
<tr>
<td>MENU( ) Function</td>
</tr>
<tr>
<td>MODIFY CLASS Command</td>
</tr>
<tr>
<td>MODIFY DATABASE Command</td>
</tr>
<tr>
<td>MODIFY GENERAL Command</td>
</tr>
<tr>
<td>MODIFY MENU Command</td>
</tr>
<tr>
<td>MODIFY QUERY Command</td>
</tr>
<tr>
<td>MODIFY STRUCTURE Command</td>
</tr>
<tr>
<td>MOUSE Command</td>
</tr>
<tr>
<td>MRKBAR( ) Function</td>
</tr>
<tr>
<td>MWINDOW( ) Function</td>
</tr>
<tr>
<td>NUMLOCK( ) Function</td>
</tr>
<tr>
<td>OBJNUM( ) Function</td>
</tr>
<tr>
<td>OEMTOANSI( ) Function</td>
</tr>
<tr>
<td>ON ESCAPE Command</td>
</tr>
<tr>
<td>ON EXIT PAD Command</td>
</tr>
<tr>
<td>ON KEY LABEL Command</td>
</tr>
<tr>
<td>ON PAGE Command</td>
</tr>
<tr>
<td>ON SELECTION MENU Command</td>
</tr>
<tr>
<td>ON SHUTDOWN Command</td>
</tr>
<tr>
<td>PACK DATABASE Command</td>
</tr>
<tr>
<td>PAD( ) Function</td>
</tr>
</tbody>
</table>

| N                                      |                             |                             |
| NUMLOCK( ) Function                   |                             |                              |

| O                                      |                             |                             |
| OBJNUM( ) Function                     | OBJTOCLIENT( ) Function     | ON BAR Command               |
| OEMTOANSI( ) Function                  | ON APLABOUT Command         | ON EXIT MENU Command         |
| ON ESCAPE Command                      | ON EXIT BAR Command         | ON KEY = Command             |
| ON EXIT PAD Command                    | ON EXIT POPUP Command       | ON PAD Command               |
| ON KEY LABEL Command                   | ON MACHELP Command          | ON SELECTION BAR Command     |
| ON PAGE Command                        | ON READERROR Command        | ON SELECTION POPUP Command   |
| ON SELECTION MENU Command              | ON SELECTION PAD Command    |                              |
| ON SHUTDOWN Command                    | OBJVAR( ) Function          |                              |

| P                                      |                             |                             |
| _PADVANCE System Memory Variable       | _PAGENO System Memory Variable | _PBPAGE System Memory Variable |
| _PCOLNO System Memory Variable         | _PCOPIES System Memory Variable | _PDRIVER System Memory Variable |
| _PDSETUP System Memory Variable        | _PECODE System Memory Variable | _PEJECT System Memory Variable |
| _PEPAGE System Memory Variable         | _PLENGTH System Memory Variable | _PLINENO System Memory Variable |
| _PLOFFSET System Memory Variable       | _PPITCH System Memory Variable | _PQUALITY System Memory Variable |
| _PRETEXT System Memory Variable        | _PSCODE System Memory Variable | _PSPACING System Memory Variable |
| _PWAIT System Memory Variable          | PACK DATABASE Command       | PACK DATABASE Command        |
| PACK DATABASE Command                  | PACK DATABASE Command       | PACK DATABASE Command        |

<p>| Other modules                           |                             |                             |
|维护菜单（MCOL）功能                  | MD命令                      | 菜单到命令（MENU TO Command）|
|内存（MEMORY）功能                    | 菜单命令（MENU Command）     | MKDIR命令（MKDIR Command）    |
|菜单（MENU）功能                       | 消息框（MESSAGEBOX）功能    | 修改连接命令（MODIFY CONNECTION Command）|
|修改类（MODIFY CLASS）命令             | 修改命令（MODIFY COMMAND Command）| 修改形式命令（MODIFY FORM Command）|
|修改数据库（MODIFY DATABASE）命令       | 修改文件（MODIFY FILE Command）| 修改备忘录命令（MODIFY MEMO Command）|
|修改一般（MODIFY GENERAL）命令          | 修改标签（MODIFY LABEL Command）| 修改项目命令（MODIFY PROJECT Command）|
|修改菜单（MODIFY MENU）命令            | 修改程序（MODIFY PROCEDURE Command）| 修改屏幕命令（MODIFY SCREEN Command）|
|修改查询（MODIFY QUERY）命令            | 修改报告（MODIFY REPORT Command）| 修改窗口命令（MODIFY WINDOW Command）|
|修改结构（MODIFY STRUCTURE）命令        | 修改视图（MODIFY VIEW Command）| 移动窗口命令（MOVE WINDOW Command）|
|鼠标（MOUSE）命令                      | 移动弹出（MOVE POPUP Command）| MROW（）函数（MROW（） Function）|
|标记栏（MRKBAR）函数                   | 标记Pad（MRKPAD）函数      |                              |
|窗口（MWINDOW）函数                    | MDOWN（）函数（MDOWN（） Function）|                              |</p>
<table>
<thead>
<tr>
<th>PCOL( ) Function</th>
<th>PEMSTATUS( ) Function</th>
<th>PLAY MACRO Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP KEY Command</td>
<td>POP MENU Command</td>
<td>POP POPUP Command</td>
</tr>
<tr>
<td>POPUP( ) Function</td>
<td>PRINTJOB ... ENDPRINTJOB Command</td>
<td>PRINTSTATUS( ) Function</td>
</tr>
<tr>
<td>PRMBAR( ) Function</td>
<td>PRMPAD( ) Function</td>
<td>PROMPT( ) Function</td>
</tr>
<tr>
<td>PROW( ) Function</td>
<td>PRINFO( ) Function</td>
<td>PUSH KEY Command</td>
</tr>
<tr>
<td>PUSH MENU Command</td>
<td>PUSH POPUP Command</td>
<td>PUTFILE( ) Function</td>
</tr>
</tbody>
</table>

Q

| QUIT Command |

R

<table>
<thead>
<tr>
<th>_RMARGIN System Memory Variable</th>
<th>RD Command</th>
<th>READKEY( ) Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ Command</td>
<td>READ MENU Command</td>
<td>RELEASE BAR Command</td>
</tr>
<tr>
<td>REFRESH() Function</td>
<td>REINDEX Command</td>
<td>RELEASE LIBRARY Command</td>
</tr>
<tr>
<td>RELEASE CLASSLIB Command</td>
<td>RELEASE Command</td>
<td>RELEASE PAD Command</td>
</tr>
<tr>
<td>RELEASE MENUS Command</td>
<td>RELEASE MODULE Command</td>
<td>RELEASE WINDOWS Command</td>
</tr>
<tr>
<td>RELEASE POPUPS Command</td>
<td>RELEASE PROCEDURE Command</td>
<td>RENAME Command</td>
</tr>
<tr>
<td>REMOVE CLASS Command</td>
<td>RENAME CLASS Command</td>
<td>RENAME VIEW Command</td>
</tr>
<tr>
<td>RENAME CONNECTION Command</td>
<td>RENAME TABLE Command</td>
<td>RESTORE FROM Command</td>
</tr>
<tr>
<td>REPORT Command</td>
<td>REQUERY( ) Function</td>
<td>RESTORE WINDOW Command</td>
</tr>
<tr>
<td>RESTORE MACROS Command</td>
<td>RESTORE SCREEN Command</td>
<td>RGBSCHEME( ) Function</td>
</tr>
<tr>
<td>RESUME Command</td>
<td>RGB( ) Function</td>
<td>RUN</td>
</tr>
<tr>
<td>RMDIR Command</td>
<td>ROW( ) Function</td>
<td></td>
</tr>
<tr>
<td>RUNSCRIPT Command</td>
<td>RDLEVEL( ) Function</td>
<td></td>
</tr>
</tbody>
</table>

S

<table>
<thead>
<tr>
<th>SAVE MACROS Command</th>
<th>SAVE SCREEN Command</th>
<th>SAVE TO Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE WINDOWS Command</td>
<td>SCHEME( ) Function</td>
<td>SCOLS( ) Function</td>
</tr>
<tr>
<td>SCROLL Command</td>
<td>_SCREEN System Memory Variable</td>
<td>SET Command</td>
</tr>
<tr>
<td>SET ALTERNATE Command</td>
<td>SET ANSI Command</td>
<td>SET APLABOUT Command</td>
</tr>
<tr>
<td>SET AUTOSAVE Command</td>
<td>SET BELL Command</td>
<td>SET BLINK Command</td>
</tr>
<tr>
<td>SET BORDER Command</td>
<td>SET BRSTATUS Command</td>
<td>SET CLASSLIB Command</td>
</tr>
<tr>
<td>SET CLEAR Command</td>
<td>SET CLOCK Command</td>
<td>SET COLOR OF Command</td>
</tr>
<tr>
<td>SET COLOR OF SCHEME Command</td>
<td>SET COLOR SET Command</td>
<td>SET COLOR TO Command</td>
</tr>
<tr>
<td>SET COMPATIBLE Command</td>
<td>SET CONFIRM Command</td>
<td>SET CONSOLE Command</td>
</tr>
<tr>
<td>SET CPCOMPILE</td>
<td>SET CPDIALOG</td>
<td>SET CURRENCY Command</td>
</tr>
<tr>
<td>Command Name</td>
<td>Command Name</td>
<td>Command Name</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>SET CURSOR</td>
<td>SET DATASESSION</td>
<td>SET DEBUG</td>
</tr>
<tr>
<td>SET DECIMALS</td>
<td>SET DELIMITERS</td>
<td>SET DEVELOPMENT</td>
</tr>
<tr>
<td>SET DEVICE</td>
<td>SET DISPLAY</td>
<td>SET DOHISTORY</td>
</tr>
<tr>
<td>SET ECHO</td>
<td>SET ESCAPE</td>
<td>SET FORMAT</td>
</tr>
<tr>
<td>SET FUNCTION</td>
<td>SET HEADINGS</td>
<td>SET HELP</td>
</tr>
<tr>
<td>SET HELPFILTER</td>
<td>SET INTENSITY</td>
<td>SET KEY</td>
</tr>
<tr>
<td>SET KEYCOMP</td>
<td>SET LOGERRORS</td>
<td>SET MACDESKTOP</td>
</tr>
<tr>
<td>SET MACHELP</td>
<td>SET MACKKEY</td>
<td>SET MARGIN</td>
</tr>
<tr>
<td>SET MARK OF</td>
<td>SET MARK TO</td>
<td>SET MEMOWIDTH</td>
</tr>
<tr>
<td>SET MESSAGE</td>
<td>SET MOUSE</td>
<td>SET ODOMETER</td>
</tr>
<tr>
<td>SET OLEOBJECT</td>
<td>SET PALETTE</td>
<td>SET POSETUP</td>
</tr>
<tr>
<td>SET POINT</td>
<td>SET PRINTER</td>
<td>SET READBORDER</td>
</tr>
<tr>
<td>SET REFRESH</td>
<td>SET RESOURCE</td>
<td>SET SAFETY</td>
</tr>
<tr>
<td>SET SCOREBOARD</td>
<td>SET SECONDS</td>
<td>SET SEPARATOR</td>
</tr>
<tr>
<td>SET SHADOWS</td>
<td>SET SKIP OF</td>
<td>SET SPACE</td>
</tr>
<tr>
<td>SET STATUS</td>
<td>SET STATUS BAR</td>
<td>SET STEP</td>
</tr>
<tr>
<td>SET STICKY</td>
<td>SET SYSFORMATS</td>
<td>SET SYSMENU</td>
</tr>
<tr>
<td>SET TALK</td>
<td>SET TEXTMERGE</td>
<td>SET TEXTMERGE DELIMITERS</td>
</tr>
<tr>
<td>SET TOPIC</td>
<td>SET TOPIC ID</td>
<td>SET TRBETWEEN</td>
</tr>
<tr>
<td>SET TYPEAHEAD</td>
<td>SET VIEW</td>
<td>SET WINDOW OF MEMO</td>
</tr>
<tr>
<td>SET XCMDFILE</td>
<td>_SHELL System Memory Variable</td>
<td>SHOW GET</td>
</tr>
<tr>
<td>SHOW GETS</td>
<td>SHOW MENU</td>
<td>SHOW OBJECT</td>
</tr>
<tr>
<td>SHOW POPUP</td>
<td>SHOW WINDOW</td>
<td>SIZE POPUP</td>
</tr>
<tr>
<td>SIZE WINDOW</td>
<td>SKPBAR( ) Function</td>
<td>SKPADD( ) Function</td>
</tr>
<tr>
<td>SOUNDEX( ) Function</td>
<td>_SPELLCHK System Memory Variable</td>
<td>SQL functions</td>
</tr>
<tr>
<td>SROWS( ) Function</td>
<td>_STARTUP System Memory Variable</td>
<td>SUSPEND Command</td>
</tr>
<tr>
<td>SYS( ) Functions except SYS(2011)</td>
<td>SYMOMETRIC( ) Function</td>
<td></td>
</tr>
</tbody>
</table>

T

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_TABS System Memory Variable</td>
<td>TEXT ... ENDTEXT Command</td>
</tr>
<tr>
<td>TRANSFORM( ) Function</td>
<td>_TRANSPORT System Memory Variable</td>
</tr>
<tr>
<td>TYPE Command</td>
<td>_THROTTLE System Memory Variable</td>
</tr>
</tbody>
</table>

U

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATED( ) Function</td>
<td>USE Command</td>
</tr>
</tbody>
</table>
Error Messages (Visual FoxPro ODBC Driver)

When an error occurs, the Visual FoxPro driver returns the following information:

- The native error number and error message text
- The SQLSTATE (an ODBC error code) and error message text

You access this error information by calling SQLError.

Native Errors

For errors that occur in the data source, the Visual FoxPro driver returns the native error number and error message text. For a list of native error numbers, see Visual FoxPro ODBC Driver Native Error Messages.

SQLSTATE (ODBC Error Codes)

For errors that are detected and returned by the Visual FoxPro driver, the 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 Visual FoxPro driver returns SQLSTATE S1000 (General error).

For a list of SQLSTATE values generated by the Visual FoxPro ODBC Driver for corresponding Visual FoxPro errors, see ODBC Error Codes.

Syntax

Error messages have the following format:

```
[vendor][ODBC_component]error_message
```

The prefixes in brackets ([ ]) identify the source of the error as defined in the following table.
### Visual FoxPro ODBC Driver Native Error Messages

The following tables list error messages native to the Visual FoxPro ODBC Driver.

#### 001

<table>
<thead>
<tr>
<th>Value</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Feature is not available.</td>
</tr>
<tr>
<td>2</td>
<td>Input/output operation failure.</td>
</tr>
<tr>
<td>3</td>
<td>Free handle is not found.</td>
</tr>
<tr>
<td>5</td>
<td>Use of unallocated handle.</td>
</tr>
<tr>
<td>99</td>
<td>Procedure canceled.</td>
</tr>
</tbody>
</table>

#### 100

<table>
<thead>
<tr>
<th>Value</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Too many files open.</td>
</tr>
<tr>
<td>101</td>
<td>Cannot open file.</td>
</tr>
<tr>
<td>102</td>
<td>Cannot create file.</td>
</tr>
<tr>
<td>105</td>
<td>Error writing to file.</td>
</tr>
<tr>
<td>107</td>
<td>Invalid key length.</td>
</tr>
<tr>
<td>109</td>
<td>Record is out of range.</td>
</tr>
<tr>
<td>110</td>
<td>Record is not in index.</td>
</tr>
<tr>
<td>111</td>
<td>Invalid file descriptor.</td>
</tr>
<tr>
<td>113</td>
<td>File is not open.</td>
</tr>
<tr>
<td>114</td>
<td>Not enough disk space for value.</td>
</tr>
<tr>
<td>115</td>
<td>Invalid operation for the cursor.</td>
</tr>
<tr>
<td>118</td>
<td>Index file does not match table.</td>
</tr>
<tr>
<td>119</td>
<td>No table is open.</td>
</tr>
<tr>
<td>120</td>
<td>File does not exist.</td>
</tr>
<tr>
<td>121</td>
<td>File already exists.</td>
</tr>
<tr>
<td>122</td>
<td>Table has no index order set.</td>
</tr>
<tr>
<td>123</td>
<td>Not a table.</td>
</tr>
<tr>
<td>125</td>
<td>Index expression exceeds maximum length.</td>
</tr>
<tr>
<td>127</td>
<td>You must use a logical expression with a FOR or WHILE clause.</td>
</tr>
<tr>
<td>Line</td>
<td>Error Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------</td>
</tr>
<tr>
<td>128</td>
<td>Not a numeric expression.</td>
</tr>
<tr>
<td>129</td>
<td>Variable is not found.</td>
</tr>
<tr>
<td>132</td>
<td>File is in use.</td>
</tr>
<tr>
<td>133</td>
<td>Index does not match the table. Delete the index file and re-create the index.</td>
</tr>
<tr>
<td>135</td>
<td>End of file encountered.</td>
</tr>
<tr>
<td>136</td>
<td>Beginning of file encountered.</td>
</tr>
<tr>
<td>137</td>
<td>Alias is not found.</td>
</tr>
<tr>
<td>139</td>
<td>You must use a logical expression with FILTER.</td>
</tr>
<tr>
<td>142</td>
<td>Cyclic relation.</td>
</tr>
<tr>
<td>143</td>
<td>No fields were found to copy.</td>
</tr>
<tr>
<td>144</td>
<td>The LOCATE command must be issued before the CONTINUE command.</td>
</tr>
<tr>
<td>145</td>
<td>Must be a character or numeric key field.</td>
</tr>
<tr>
<td>146</td>
<td>Cannot write to a read-only file.</td>
</tr>
<tr>
<td>147</td>
<td>Target table is already engaged in a relation.</td>
</tr>
<tr>
<td>148</td>
<td>Expression has been re-entered while the filter is executing.</td>
</tr>
<tr>
<td>149</td>
<td>Not enough memory for buffer.</td>
</tr>
<tr>
<td>150</td>
<td>Not enough memory for file map.</td>
</tr>
<tr>
<td>155</td>
<td>Invalid buffdirt call.</td>
</tr>
<tr>
<td>156</td>
<td>Duplicate field names.</td>
</tr>
<tr>
<td>158</td>
<td>No fields found to process.</td>
</tr>
<tr>
<td>159</td>
<td>Numeric overflow. Data was lost.</td>
</tr>
<tr>
<td>162</td>
<td>Procedure 'value' is not found.</td>
</tr>
<tr>
<td>165</td>
<td>'value' is not related to the current work area.</td>
</tr>
<tr>
<td>170</td>
<td>'value' is not found.</td>
</tr>
<tr>
<td>171</td>
<td>Cannot open file 'value'.</td>
</tr>
<tr>
<td>173</td>
<td>'value' does not exist.</td>
</tr>
<tr>
<td>174</td>
<td>'value' is not a memory variable.</td>
</tr>
<tr>
<td>175</td>
<td>'value' is not a file variable.</td>
</tr>
<tr>
<td>176</td>
<td>'value' is not an array.</td>
</tr>
<tr>
<td>177</td>
<td>Alias 'value' is not found.</td>
</tr>
<tr>
<td>180</td>
<td>File was not placed in memory using the LOAD command.</td>
</tr>
<tr>
<td>182</td>
<td>There is not enough memory to complete this operation.</td>
</tr>
<tr>
<td>200</td>
<td>Syntax error.</td>
</tr>
<tr>
<td>201</td>
<td>Too many names used.</td>
</tr>
<tr>
<td>202</td>
<td>Program is too large.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>203</td>
<td>Too many memory variables.</td>
</tr>
<tr>
<td>205</td>
<td>Nesting error.</td>
</tr>
<tr>
<td>206</td>
<td>Recursive macro definition.</td>
</tr>
<tr>
<td>209</td>
<td>Line is too long.</td>
</tr>
<tr>
<td>210</td>
<td>Allowed DO nesting level exceeded.</td>
</tr>
<tr>
<td>211</td>
<td>An IF</td>
</tr>
<tr>
<td>212</td>
<td>Structure nesting is too deep.</td>
</tr>
<tr>
<td>213</td>
<td>There is a missing keyword in the FOR...ENDFOR or DO CASE...ENDCASE command structure.</td>
</tr>
<tr>
<td>219</td>
<td>Command contains unrecognized phrase/keyword.</td>
</tr>
<tr>
<td>221</td>
<td>Command is missing required clause.</td>
</tr>
<tr>
<td>222</td>
<td>Unrecognized command verb.</td>
</tr>
<tr>
<td>224</td>
<td>Invalid subscript reference.</td>
</tr>
<tr>
<td>227</td>
<td>Missing expression.</td>
</tr>
<tr>
<td>228</td>
<td>Table number is invalid.</td>
</tr>
<tr>
<td>229</td>
<td>Too few arguments.</td>
</tr>
<tr>
<td>230</td>
<td>Too many arguments.</td>
</tr>
<tr>
<td>233</td>
<td>Statement is not allowed in interactive mode.</td>
</tr>
<tr>
<td>234</td>
<td>Subscript is outside defined range.</td>
</tr>
<tr>
<td>236</td>
<td>Suspend program before using RESUME.</td>
</tr>
<tr>
<td>238</td>
<td>No PARAMETER statement is found.</td>
</tr>
<tr>
<td>239</td>
<td>Must specify additional parameters.</td>
</tr>
<tr>
<td>240</td>
<td>Not a character expression.</td>
</tr>
<tr>
<td>250</td>
<td>Too many PROCEDURE commands are in effect.</td>
</tr>
<tr>
<td>252</td>
<td>Compiled code for this line is too long.</td>
</tr>
<tr>
<td>257</td>
<td>Key string is too long.</td>
</tr>
<tr>
<td>291</td>
<td>Expression used with ASIN() is out of range.</td>
</tr>
<tr>
<td>292</td>
<td>Cannot use 0 or negative as the argument for LOG10().</td>
</tr>
<tr>
<td>293</td>
<td>Expression used with ACOS() is out of range.</td>
</tr>
<tr>
<td>294</td>
<td>FOXUSER.DBF file is invalid.</td>
</tr>
<tr>
<td>295</td>
<td>Invalid path or file name.</td>
</tr>
<tr>
<td>296</td>
<td>Error reading the resource.</td>
</tr>
<tr>
<td>297</td>
<td>Command is allowed only in interactive mode.</td>
</tr>
<tr>
<td>300</td>
<td>Operator/operand type mismatch.</td>
</tr>
<tr>
<td>301</td>
<td>Data type mismatch.</td>
</tr>
<tr>
<td>305</td>
<td>Expression evaluated to an illegal value.</td>
</tr>
<tr>
<td>Code</td>
<td>Message</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td>307</td>
<td>Cannot divide by 0.</td>
</tr>
<tr>
<td>308</td>
<td>Insufficient stack space.</td>
</tr>
<tr>
<td>337</td>
<td>Cannot nest the PRINTJOB command.</td>
</tr>
</tbody>
</table>

### 400

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>406</td>
<td>Printer is not ready.</td>
</tr>
<tr>
<td>407</td>
<td>Invalid argument used with the SET function.</td>
</tr>
<tr>
<td>410</td>
<td>Unable to create temporary work files.</td>
</tr>
<tr>
<td>423</td>
<td>Error creating the OLE object.</td>
</tr>
<tr>
<td>424</td>
<td>Error copying the OLE object to the Clipboard.</td>
</tr>
<tr>
<td>462</td>
<td>Internal consistency error.</td>
</tr>
<tr>
<td>465</td>
<td>SQL pass-through internal consistency error.</td>
</tr>
<tr>
<td>466</td>
<td>Connection handle is invalid.</td>
</tr>
<tr>
<td>467</td>
<td>Property is invalid for local cursors.</td>
</tr>
<tr>
<td>468</td>
<td>Property is invalid for table cursors.</td>
</tr>
<tr>
<td>469</td>
<td>Property value is out of bounds.</td>
</tr>
<tr>
<td>470</td>
<td>Incorrect property name.</td>
</tr>
<tr>
<td>471</td>
<td>Incorrect column format.</td>
</tr>
<tr>
<td>473</td>
<td>Environment-level property is invalid.</td>
</tr>
<tr>
<td>474</td>
<td>Invalid call issued while executing a SQLEXEC() sequence.</td>
</tr>
<tr>
<td>479</td>
<td>Invalid update column name.</td>
</tr>
<tr>
<td>489</td>
<td>General fields cannot be used in the WHERE condition of an update statement. Change the WhereType property of the view.</td>
</tr>
<tr>
<td>491</td>
<td>No update tables are specified. Use the Tables property of the cursor.</td>
</tr>
<tr>
<td>492</td>
<td>No key columns are specified for the update table. Use the KeyFieldList property of the cursor.</td>
</tr>
<tr>
<td>493</td>
<td>SQL parameter is missing.</td>
</tr>
<tr>
<td>494</td>
<td>View definition has been changed.</td>
</tr>
<tr>
<td>495</td>
<td>Warning: The key defined by the KeyField property for table value is not unique.</td>
</tr>
<tr>
<td>498</td>
<td>SQL SELECT statement is invalid.</td>
</tr>
<tr>
<td>499</td>
<td>SQL parameter value is invalid.</td>
</tr>
</tbody>
</table>

### 500

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>502</td>
<td>Cannot write to the record because it is in use.</td>
</tr>
<tr>
<td>503</td>
<td>File cannot be locked.</td>
</tr>
<tr>
<td>508</td>
<td>Error initializing OLE.</td>
</tr>
<tr>
<td>520</td>
<td>No database is open or set as the current database.</td>
</tr>
<tr>
<td>522</td>
<td>Connectivity internal consistency error.</td>
</tr>
<tr>
<td>Line</td>
<td>Message</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td>523</td>
<td>Execution was canceled by the user.</td>
</tr>
<tr>
<td>525</td>
<td>Function is not supported on remote tables.</td>
</tr>
<tr>
<td>526</td>
<td>Connectivity error: value.</td>
</tr>
<tr>
<td>527</td>
<td>Cannot load ODBC library, ODBC32.DLL.</td>
</tr>
<tr>
<td>528</td>
<td>ODBC entry point missing, value.</td>
</tr>
<tr>
<td>530</td>
<td>Fetching canceled; remote table is closed.</td>
</tr>
<tr>
<td>532</td>
<td>Type conversion is not supported.</td>
</tr>
<tr>
<td>533</td>
<td>This property is read-only.</td>
</tr>
<tr>
<td>536</td>
<td>Function is not supported on native tables.</td>
</tr>
<tr>
<td>538</td>
<td>A stored procedure is executing.</td>
</tr>
<tr>
<td>540</td>
<td>Session number is invalid.</td>
</tr>
<tr>
<td>541</td>
<td>Connection value is busy.</td>
</tr>
<tr>
<td>542</td>
<td>Base table fields have been changed and no longer match view fields. View field properties cannot be set.</td>
</tr>
<tr>
<td>543</td>
<td>Type conversion required by the DataType property for field 'value' is invalid.</td>
</tr>
<tr>
<td>544</td>
<td>DataType property for field 'value' is invalid.</td>
</tr>
<tr>
<td>545</td>
<td>Table buffer for alias 'value' contains uncommitted changes.</td>
</tr>
<tr>
<td>546</td>
<td>Cannot close table during execution of table-bound expression.</td>
</tr>
<tr>
<td>547</td>
<td>Cannot insert an empty row from a view into its base table(s).</td>
</tr>
<tr>
<td>548</td>
<td>Table value has one or more non-structural indexes open. Please close them and retry the Begin Transaction.</td>
</tr>
<tr>
<td>549</td>
<td>Data session #value cannot be released with open transaction(s).</td>
</tr>
<tr>
<td>550</td>
<td>.DBC internal consistency error.</td>
</tr>
<tr>
<td>557</td>
<td>The database must be opened exclusively.</td>
</tr>
<tr>
<td>559</td>
<td>Property is not found.</td>
</tr>
<tr>
<td>560</td>
<td>Property value is invalid.</td>
</tr>
<tr>
<td>561</td>
<td>Database is invalid. Please validate.</td>
</tr>
<tr>
<td>562</td>
<td>Cannot find object value in the database.</td>
</tr>
<tr>
<td>563</td>
<td>Cannot find view value in the current database.</td>
</tr>
<tr>
<td>566</td>
<td>Cannot issue the PACK command on a database while its tables are in use.</td>
</tr>
<tr>
<td>567</td>
<td>Primary key property is invalid; please validate database.</td>
</tr>
<tr>
<td>570</td>
<td>Database is read-only.</td>
</tr>
<tr>
<td>571</td>
<td>The name value is already used for another</td>
</tr>
<tr>
<td>575</td>
<td>Object name is invalid.</td>
</tr>
<tr>
<td>577</td>
<td>Table value is referenced in a relation.</td>
</tr>
<tr>
<td>578</td>
<td>Invalid database table name.</td>
</tr>
<tr>
<td>579</td>
<td>Command cannot be issued on a table with cursors in table buffering mode.</td>
</tr>
<tr>
<td>580</td>
<td>Feature is not supported for non-.DBC tables.</td>
</tr>
<tr>
<td>581</td>
<td>Field value does not accept null value.</td>
</tr>
<tr>
<td>583</td>
<td>Record validation rule is violated.</td>
</tr>
<tr>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>585</td>
<td>Update conflict. Use TABLEUPDATE() with the lForce parameter to commit the update or TABLEREVERT() to roll back the update.</td>
</tr>
<tr>
<td>586</td>
<td>Function requires row or table buffering mode.</td>
</tr>
<tr>
<td>587</td>
<td>Illegal nested OLDVAL() or CURVAL().</td>
</tr>
<tr>
<td>589</td>
<td>Table or row buffering requires that SET MULTILOCKS is set to ON.</td>
</tr>
<tr>
<td>590</td>
<td>BEGIN TRANSACTION command failed. Nesting level is too deep.</td>
</tr>
<tr>
<td>591</td>
<td>END TRANSACTION command cannot be issued without a corresponding BEGIN TRANSACTION command.</td>
</tr>
<tr>
<td>592</td>
<td>ROLLBACK command cannot be issued without a corresponding BEGIN TRANSACTION command.</td>
</tr>
<tr>
<td>593</td>
<td>Command cannot be issued within a transaction.</td>
</tr>
<tr>
<td>594</td>
<td>Illegal to attempt a file lock in a transaction after taking prior record locks.</td>
</tr>
<tr>
<td>596</td>
<td>Table buffering is not enabled.</td>
</tr>
<tr>
<td>597</td>
<td>Views require either DB_BUFOPTROW or DB_BUFOPTTABLE.</td>
</tr>
<tr>
<td>598</td>
<td>Rule and trigger code must balance transaction usage.</td>
</tr>
<tr>
<td>599</td>
<td>Data session #value was forced to ROLLBACK all transactions to avoid deadlock.</td>
</tr>
<tr>
<td>600</td>
<td>Alias name is already in use.</td>
</tr>
<tr>
<td>601</td>
<td>Operation is invalid for a Memo, General, or Picture field.</td>
</tr>
<tr>
<td>612</td>
<td>No such menu or menu item is defined.</td>
</tr>
<tr>
<td>618</td>
<td>Menu has not been defined with DEFINE MENU.</td>
</tr>
<tr>
<td>624</td>
<td>Menu title has not been defined with DEFINE PAD.</td>
</tr>
<tr>
<td>625</td>
<td>Menu has not been defined with DEFINE POPUP.</td>
</tr>
<tr>
<td>631</td>
<td>Array dimensions are invalid.</td>
</tr>
<tr>
<td>637</td>
<td>File must be opened exclusively to convert the Memo file.</td>
</tr>
<tr>
<td>638</td>
<td>Field must be a Memo field.</td>
</tr>
<tr>
<td>649</td>
<td>No previous PRINTJOB command to correspond to this command.</td>
</tr>
<tr>
<td>651</td>
<td>CANCEL or SUSPEND is not allowed.</td>
</tr>
<tr>
<td>659</td>
<td>The table has memo fields that cannot be converted while open read-only.</td>
</tr>
<tr>
<td>683</td>
<td>Index tag is not found.</td>
</tr>
<tr>
<td>700</td>
<td>Record is in use by another user.</td>
</tr>
<tr>
<td>701</td>
<td>File must be opened exclusively.</td>
</tr>
<tr>
<td>702</td>
<td>File is in use by another user.</td>
</tr>
<tr>
<td>703</td>
<td>Record is not locked.</td>
</tr>
<tr>
<td>705</td>
<td>File access is denied.</td>
</tr>
<tr>
<td>706</td>
<td>Cannot sort .IDX files in descending order.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>707</td>
<td>Structural .CDX file is not found.</td>
</tr>
<tr>
<td>708</td>
<td>File is open in another work area.</td>
</tr>
<tr>
<td>712</td>
<td>Field name is a duplicate or invalid.</td>
</tr>
<tr>
<td>714</td>
<td>Window 'value' has not been defined.</td>
</tr>
<tr>
<td>718</td>
<td>File is read-only.</td>
</tr>
<tr>
<td>722</td>
<td>Preprocessor expression is invalid.</td>
</tr>
<tr>
<td>734</td>
<td>Property value is not found.</td>
</tr>
<tr>
<td>737</td>
<td>value is a method, event, or object.</td>
</tr>
<tr>
<td>738</td>
<td>Property value is not a method or event.</td>
</tr>
<tr>
<td>740</td>
<td>value is a read-only property.</td>
</tr>
<tr>
<td>748</td>
<td>This file is incompatible with the current version of Visual FoxPro.</td>
</tr>
<tr>
<td>750</td>
<td>File was created in a later version of Visual FoxPro than the current version.</td>
</tr>
<tr>
<td>763</td>
<td>Property value already exists.</td>
</tr>
<tr>
<td>773</td>
<td>Database object type is invalid.</td>
</tr>
<tr>
<td>784</td>
<td>This object is derived from a base class and does not have a parent class.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>SQL: Cannot locate table.</td>
</tr>
<tr>
<td>802</td>
<td>Too many columns.</td>
</tr>
<tr>
<td>872</td>
<td>No primary key.</td>
</tr>
<tr>
<td>879</td>
<td>Uniqueness of index value is violated.</td>
</tr>
<tr>
<td>884</td>
<td>Only structural tags can be defined as candidate.</td>
</tr>
<tr>
<td>885</td>
<td>Index does not accept NULL.</td>
</tr>
<tr>
<td>886</td>
<td>Illegal recursion in rule evaluation.</td>
</tr>
<tr>
<td>887</td>
<td>Tag name is too long.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>900</td>
<td>Function argument value, type, or count is invalid.</td>
</tr>
<tr>
<td>901</td>
<td>Expression evaluator failed.</td>
</tr>
<tr>
<td>903</td>
<td>String is too long to fit.</td>
</tr>
<tr>
<td>904</td>
<td>** or ^ domain error.</td>
</tr>
<tr>
<td>905</td>
<td>LOG(): Zero or negative used as argument.</td>
</tr>
<tr>
<td>906</td>
<td>SQRT() argument cannot be negative.</td>
</tr>
<tr>
<td>912</td>
<td>Operation is invalid for a General field.</td>
</tr>
<tr>
<td>914</td>
<td>Code page number is invalid.</td>
</tr>
<tr>
<td>915</td>
<td>Collating sequence 'value' is not found.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Message</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>918</td>
<td>File name is too long.</td>
</tr>
<tr>
<td>922</td>
<td>Volume does not exist.</td>
</tr>
<tr>
<td>923</td>
<td>Object value is not found.</td>
</tr>
<tr>
<td>924</td>
<td>value is not an object.</td>
</tr>
<tr>
<td>925</td>
<td>Unknown member value.</td>
</tr>
<tr>
<td>928</td>
<td>Statement is only valid within a class definition.</td>
</tr>
<tr>
<td>929</td>
<td>value can only be used within a method.</td>
</tr>
<tr>
<td>930</td>
<td>Cannot redefine value.</td>
</tr>
<tr>
<td>931</td>
<td>Statement is not in a procedure.</td>
</tr>
<tr>
<td>934</td>
<td>Statement is only valid within a method.</td>
</tr>
<tr>
<td>935</td>
<td>The current object does not inherit from class value.</td>
</tr>
<tr>
<td>937</td>
<td>Procedure file 'value' is not found.</td>
</tr>
<tr>
<td>938</td>
<td>Object is not contained in a value.</td>
</tr>
<tr>
<td>939</td>
<td>WITH/ENDWITH mismatch.</td>
</tr>
<tr>
<td>940</td>
<td>Expression is not valid outside of WITH/ENDWITH.</td>
</tr>
<tr>
<td>941</td>
<td>Error code is invalid.</td>
</tr>
<tr>
<td>942</td>
<td>Objects cannot be assigned to arrays.</td>
</tr>
<tr>
<td>943</td>
<td>Member value does not evaluate to an object.</td>
</tr>
<tr>
<td>945</td>
<td>The current object has been released.</td>
</tr>
<tr>
<td>947</td>
<td>Expression is too complex.</td>
</tr>
<tr>
<td>951</td>
<td>Cannot clear the object in use.</td>
</tr>
<tr>
<td>955</td>
<td>WIN.INI/registry is corrupted.</td>
</tr>
<tr>
<td>957</td>
<td>Error accessing printer spooler.</td>
</tr>
<tr>
<td>959</td>
<td>Invalid coordinates.</td>
</tr>
<tr>
<td>960</td>
<td>Illegal redefinition of variable value.</td>
</tr>
<tr>
<td>971</td>
<td>Cannot compile until the current COMPILE command has completed.</td>
</tr>
<tr>
<td>972</td>
<td>Array value is in use.</td>
</tr>
<tr>
<td>974</td>
<td>Arrays cannot be assigned to array elements.</td>
</tr>
<tr>
<td>976</td>
<td>Cannot resolve backlink.</td>
</tr>
<tr>
<td>988</td>
<td>Currency value is out of range.</td>
</tr>
<tr>
<td>990</td>
<td>Cancel.</td>
</tr>
<tr>
<td>999</td>
<td>Function is not implemented.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft

**ODBC Error Codes (Visual FoxPro ODBC Driver)**

The following table lists Visual FoxPro error codes mapped to ODBC Error Code SQLSTATE values. The mapped SQLSTATE values come from SQLExecDirect and SQLPrepare. No other SQLSTATE values from other ODBC API are mapped because SQLExecDirect and SQLPrepare are the only functions that access the Visual FoxPro engine.
For more information on ODBC error codes, see Appendix A: ODBC Error Codes, in the ODBC Programmer’s Reference.

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Visual FoxPro error code</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1001</td>
<td>149 150 182 202 308</td>
</tr>
<tr>
<td>1004</td>
<td>159</td>
</tr>
<tr>
<td>37000</td>
<td>132 200 219 221 222 227 229 230 498 499 713 901</td>
</tr>
<tr>
<td>22005</td>
<td>301 302</td>
</tr>
<tr>
<td>22012</td>
<td>307</td>
</tr>
<tr>
<td>23000</td>
<td>581 583 884 886 988</td>
</tr>
<tr>
<td>S0001</td>
<td>121 571</td>
</tr>
<tr>
<td>S0002</td>
<td>173 120 123 295 562 563 802</td>
</tr>
<tr>
<td>S0012</td>
<td>683</td>
</tr>
<tr>
<td>S0021</td>
<td>156 712</td>
</tr>
<tr>
<td>S0022</td>
<td>158</td>
</tr>
<tr>
<td>S1000</td>
<td>806</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td></td>
</tr>
<tr>
<td>135</td>
<td></td>
</tr>
<tr>
<td>136</td>
<td></td>
</tr>
<tr>
<td>137</td>
<td></td>
</tr>
<tr>
<td>145</td>
<td></td>
</tr>
<tr>
<td>146</td>
<td></td>
</tr>
<tr>
<td>171</td>
<td></td>
</tr>
<tr>
<td>173</td>
<td></td>
</tr>
<tr>
<td>177</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td></td>
</tr>
<tr>
<td>205</td>
<td></td>
</tr>
<tr>
<td>239</td>
<td></td>
</tr>
<tr>
<td>240</td>
<td></td>
</tr>
<tr>
<td>252</td>
<td></td>
</tr>
<tr>
<td>257</td>
<td></td>
</tr>
<tr>
<td>296</td>
<td></td>
</tr>
<tr>
<td>305</td>
<td></td>
</tr>
<tr>
<td>407</td>
<td></td>
</tr>
<tr>
<td>410</td>
<td></td>
</tr>
<tr>
<td>462</td>
<td></td>
</tr>
<tr>
<td>502</td>
<td></td>
</tr>
<tr>
<td>503</td>
<td></td>
</tr>
<tr>
<td>520</td>
<td></td>
</tr>
<tr>
<td>538</td>
<td></td>
</tr>
<tr>
<td>550</td>
<td></td>
</tr>
<tr>
<td>561</td>
<td></td>
</tr>
<tr>
<td>567</td>
<td></td>
</tr>
<tr>
<td>570</td>
<td></td>
</tr>
<tr>
<td>575</td>
<td></td>
</tr>
</tbody>
</table>
Troubleshooting (Visual FoxPro ODBC Driver)

The following sections discuss how to improve performance and solve problems you might encounter while using the Visual FoxPro ODBC Driver.

Accessing Parameterized Views

You can't access parameterized views in a Visual FoxPro database using the driver. A parameterized view creates a WHERE clause in the view’s SQL SELECT statement that limits the records downloaded to those records that meet the conditions of the WHERE clause built using the value supplied for the parameter. Because the driver doesn’t support passing parameters to the view, attempts to access a parameterized view by using the driver will fail.

The parameter value can be supplied at run time or passed programmatically to the view.

Accessing Remote Views

You can't access remote views in a Visual FoxPro database using the driver. Remote views are views that access either non-FoxPro data or a combination of FoxPro and non-FoxPro data. To access remote views, use Visual FoxPro.

Deleting Records

You can mark records for deletion using the driver, but you can't permanently remove records from the database. To permanently remove records from a table, use Visual FoxPro.

Increasing Performance Using Background Fetching

You can improve performance on large fetches by using the background fetching feature of the driver. Background fetching uses a separate thread to fetch
data requested from a specific data source.

You can employ background fetching for a data source in one of the following ways:

- Check the **Fetch data in background** checkbox on the **ODBC Visual FoxPro Setup** dialog box.
- Use the `BackgroundFetch` attribute keyword in your connection string.

For information on connection string attribute keywords, see **Using Connection Strings**.

### Updating Multitiered Views

A multitiered view is a view based on one or more views rather than on a base table. When you update data in a multitiered view, the updates go down only one level, to the view on which the top-level view is based; base tables are not updated.

### Using Data Definition Language (DDL) in Stored Procedures

You can't use DDL, such as `CREATE TABLE` or `ALTER TABLE`, in Visual FoxPro stored procedures.

For information on language you can use in stored procedures, see **Support for Rules, Triggers, Default Values, and Stored Procedures**.

### Using Positioned Updates

The driver doesn't support positioned updates. Use the SQL `WHERE` clause to identify the rows you want to update.

### Using the SET ANSI Command

If you're a Visual FoxPro developer, you should be aware that the default setting for SET ANSI is ON for the driver, in contrast to a default setting of OFF for Visual FoxPro. The default ON setting for SET ANSI allows Visual FoxPro data sources to behave consistently with other ODBC data sources that typically perform exact comparisons. You can change the default setting. For more information, see **SET ANSI**.

---

## ODBC Functions and the Visual FoxPro ODBC Driver

The topics in this section provide a brief summary of ODBC API functions and any Visual FoxPro–specific details.

### Core Level API Support

The ODBC Core Level API functions are listed in the following table. All of these functions are supported by the Visual FoxPro ODBC Driver.

<table>
<thead>
<tr>
<th>Core Level API Functions</th>
<th>Visual FoxPro ODBC Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocConnect</td>
<td>SQLExecute</td>
</tr>
<tr>
<td>SQLAllocEnv</td>
<td>SQLFetch</td>
</tr>
<tr>
<td>SQLAllocStmt</td>
<td>SQLFreeConnect</td>
</tr>
<tr>
<td>SQLBindCol</td>
<td>SQLFreeEnv</td>
</tr>
<tr>
<td>SQLCancel</td>
<td>SQLFreeStmt</td>
</tr>
<tr>
<td>SQLColAttributes</td>
<td>SQLGetCursorName</td>
</tr>
</tbody>
</table>
Level 1 API Support

The ODBC Level 1 API functions are listed in the following table. All of these functions are fully or partially supported by the Visual FoxPro ODBC Driver.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLConnect</td>
<td>SQLNumResultCols</td>
</tr>
<tr>
<td>SQLDescribeCol</td>
<td>SQLPrepare</td>
</tr>
<tr>
<td>SQLDisconnect</td>
<td>SQLRowCount</td>
</tr>
<tr>
<td>SQLError</td>
<td>SQLSetCursorName</td>
</tr>
<tr>
<td>SQLExecDirect</td>
<td>SQLTransact</td>
</tr>
<tr>
<td>SQLBindParameter</td>
<td>SQLGetTypeInfo</td>
</tr>
<tr>
<td>SQLColumns</td>
<td>SQLParamData</td>
</tr>
<tr>
<td>SQLDriverConnect</td>
<td>SQLPutData</td>
</tr>
<tr>
<td>SQLGetConnectOption</td>
<td>SQLSetConnectOption</td>
</tr>
<tr>
<td>SQLGetData</td>
<td>SQLSetStmtOption</td>
</tr>
<tr>
<td>SQLGetFunctions</td>
<td>SQLSpecialColumns</td>
</tr>
<tr>
<td>SQLGetInfo</td>
<td>SQLStatistics</td>
</tr>
<tr>
<td>SQLGetStmtOption</td>
<td>SQLTables</td>
</tr>
</tbody>
</table>

Level 2 API Support

The following ODBC Level 2 API functions are fully or partially supported:

- SQLDataSources
- SQLDrivers
- SQLExtendedFetch
- SQLMoreResults
- SQLNumParams
- SQLParamOptions
- SQLPrimaryKeys
- SQLSetPos
- SQLSetScrollOptions (partial support)

The following Level 2 API functions are not supported:

- SQLBrowseConnect
- SQLColumnPrivileges
- SQLDescribeParam
- SQLForeignKeys
- SQLNativeSql
- SQLProcedureColumns
- SQLProcedures
- SQLTablePrivileges
SQLAllocConnect (Visual FoxPro ODBC Driver)

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Allocates memory for a connection handle, hdbc, within the environment identified by henv. The Driver Manager processes this call and calls the driver's SQLAllocConnect whenever SQLConnect, SQLBrowseConnect, or SQLDriverConnect is called.

For more information, see SQLAllocConnect in the ODBC Programmer's Reference.

SQLAllocEnv (Visual FoxPro ODBC Driver)

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Allocates memory for an environment handle, henv, and initializes the ODBC call level interface for use by an application.

For more information, see SQLAllocEnv in the ODBC Programmer's Reference.

SQLAllocStmt (Visual FoxPro ODBC Driver)

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Allocates memory for a statement handle and associates the statement handle with the connection specified by hdbc. The Driver Manager passes this call to the driver, which allocates the memory for the hstmt structure.

For more information, see SQLAllocStmt in the ODBC Programmer's Reference.

SQLBindCol (Visual FoxPro ODBC Driver)

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Assigns storage space for a result column and specifies the type of the result. When SQLFetch or SQLExtendedFetch is called, the driver places the data for all bound columns in the assigned locations. See SQLGetTypeInfo for the mapping between ODBC and Visual FoxPro data types.

For more information, see SQLBindCol in the ODBC Programmer's Reference.
SQLBindParameter (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 1

Associates a buffer with a parameter marker in an SQL statement. The Visual FoxPro ODBC Driver supports input parameters as specified by the `fParamType` argument.

For more information, see `SQLBindParameter` in the ODBC Programmer’s Reference.

SQLCancel (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Cancels the processing on a statement handle, `hstmt`.

For more information, see `SQLCancel` in the ODBC Programmer’s Reference.

SQLColAttributes (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Returns descriptor information for a column in a result set. Descriptor information is returned as a character string, a 32-bit descriptor-dependent value, or an integer value.

**Note**

`SQLColAttributes` cannot be used to return information about the bookmark column (column 0).

The Visual FoxPro ODBC Driver supports all `fDescType` values. The following table includes comments on the driver’s implementation of selected values.

<table>
<thead>
<tr>
<th><code>fDescType</code></th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_COLUMN_AUTO_INCREMENT</td>
<td>Returns FALSE: Visual FoxPro has no counter fields.</td>
</tr>
<tr>
<td>SQL_COLUMN_CASE_SENSITIVE</td>
<td>Always returns TRUE if the column type is Character.</td>
</tr>
<tr>
<td>SQL_COLUMN_LABEL</td>
<td>Returns the column name, which is also returned by SQL_COLUMN_NAME.</td>
</tr>
<tr>
<td>SQL_COLUMN_MONEY</td>
<td>Returns TRUE if the column type is Currency (represented by a &quot;Y&quot; in the Visual FoxPro language).</td>
</tr>
<tr>
<td>SQL_COLUMN_OWNER_NAME</td>
<td>Always returns an empty string.</td>
</tr>
</tbody>
</table>
### SQLColumns (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

**Support:** Full

**ODBC API Conformance:** Level 1

For a table, creates a result set that is the column list for the specified table or tables.

For more information, see SQLColumns in the ODBC Programmer’s Reference.

### SQLConnect (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

**Support:** Full

**ODBC API Conformance:** Core Level

Connects to a data source, which can be either a database or a directory of tables. The Visual FoxPro ODBC Driver ignores the szUID, cbUID, szAuthStr, and cbAuthStr arguments.

For more information, see SQLConnect in the ODBC Programmer’s Reference.

### SQLDataSources (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

**Support:** Full

**ODBC API Conformance:** Level 2

Lists data source names.

For more information, see SQLDataSources in the ODBC Programmer’s Reference.
SQLDisconnect (Visual FoxPro ODBC Driver)

Closes a connection.

For more information, see SQLDisconnect in the ODBC Programmer's Reference.

SQLDriverConnect (Visual FoxPro ODBC Driver)

Connects to an existing data source, which can be either a database or a directory of free tables. The ODBC attribute keywords UID and PWD are ignored. The following table lists the additional supported attribute keywords.

<table>
<thead>
<tr>
<th>ODBC attribute keyword</th>
<th>Attribute value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN</td>
<td></td>
</tr>
<tr>
<td>UID</td>
<td>Ignored by the Visual FoxPro ODBC Driver but does not generate an error.</td>
</tr>
<tr>
<td>PWD</td>
<td>Ignored by the Visual FoxPro ODBC Driver but does not generate an error.</td>
</tr>
<tr>
<td>Driver</td>
<td>The name and location of the Visual FoxPro ODBC Driver; implemented by the Driver Manager.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Visual FoxPro ODBC Driver attribute keyword</th>
<th>Attribute value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackgroundFetch</td>
<td>&quot;Yes&quot; or &quot;No&quot;</td>
</tr>
<tr>
<td>Collate</td>
<td>&quot;Machine&quot; or other collating sequence. For a list of supported collating sequences, see SET COLLATE.</td>
</tr>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>Exclusive</td>
<td>&quot;Yes&quot; or &quot;No&quot;</td>
</tr>
<tr>
<td>SourceDB</td>
<td>A fully qualified path to a directory containing zero or more free tables, or the absolute path and file name for a database.</td>
</tr>
</tbody>
</table>
If the data source name is not specified, the Driver Manager prompts the user for the information (depending on the setting of the fDriverCompletion argument) and then continues. If more information is required, the Visual FoxPro ODBC Driver displays the prompt dialog.

For more information, see SQLDriverConnect in the ODBC Programmer's Reference.

© 2016 Microsoft

SQLDrivers (Visual FoxPro ODBC Driver)

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.</td>
</tr>
</tbody>
</table>

Support: Full

ODBC API Conformance: Level 2

Lists driver descriptions and driver attribute keywords.

For more information, see SQLDrivers in the ODBC Programmer's Reference.

© 2016 Microsoft

SQLError (Visual FoxPro ODBC Driver)

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.</td>
</tr>
</tbody>
</table>

Support: Full

ODBC API Conformance: Core Level

Returns error or status information about the last error. The driver maintains a stack or list of errors that can be returned for the hstmt, hdbc, and henv arguments, depending on how the call to SQLError is made. The error queue is flushed after each statement.

The following table describes the SQLError arguments and return values used by the driver.

<table>
<thead>
<tr>
<th>SQLError argument</th>
<th>Return value description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szSQLState</td>
<td>The value for the SQLSTATE represented by the error.</td>
</tr>
<tr>
<td>pfNativeError</td>
<td>A nonzero value indicates a Visual FoxPro ODBC Driver Native Error Message. A value of zero indicates the error has been detected by the driver and mapped to the appropriate ODBC Error Code.</td>
</tr>
<tr>
<td>szErrorMsg</td>
<td>The text for the native error or ODBC error.</td>
</tr>
<tr>
<td>pcbErrorMsg</td>
<td>The length of the message text plus the length of the identifiers.</td>
</tr>
</tbody>
</table>

For more information on driver error messages, see Error Messages Overview. For more information about this function, see SQLError in the ODBC Programmer's Reference.

© 2016 Microsoft

SQLExecDirect (Visual FoxPro ODBC Driver)

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.</td>
</tr>
</tbody>
</table>

Support: Full
Executes a new, preparable SQL statement. The Visual FoxPro ODBC Driver uses the current values of the parameter marker variables if any parameters exist in the statement.

To create a batch command to submit more than one SQL statement at a time, use a semicolon (;) to separate each SQL statement in the batch.

If your table, view, or field names contain spaces, enclose the names in back quote marks. For example, if your database contains a table named My Table and the field My Field, enclose each element of the identifier as follows:

```
SELECT 'My Table'.`Field1`, 'My Table'.`Field2` FROM 'My Table'
```

For more information, see SQLExecDirect in the ODBC Programmer's Reference.

SQLExecute (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Executes a prepared SQL statement (a statement already prepared by SQLPrepare). The driver uses the current values of the parameter marker variables if any parameters exist in the statement.

For more information, see SQLExecute in the ODBC Programmer's Reference.

SQLExtendedFetch (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 2

Similar to SQLFetch but returns multiple rows using an array for each column. The result set is forward-scrollable and can be made backward-scrollable if the cursor is defined to be static, not forward-only.

By default, the Visual FoxPro ODBC Driver does not return rows marked as deleted in a FoxPro table. Rows marked for deletion but not yet removed from a table are not included in the result set cursor. You can change this behavior by using the SET DELETED command.

For more information, see SQLExtendedFetch in the ODBC Programmer's Reference.

SQLFetch (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Retrieves one row from a result set into the locations specified by the previous calls to SQLBindCol. Prepares the driver for a call to SQLGetData for the unbound columns.

For more information, see SQLFetch in the ODBC Programmer's Reference.
SQLFreeConnect (Visual FoxPro ODBC Driver)

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.</td>
</tr>
</tbody>
</table>

Support: Full

ODBC API Conformance: Core Level

Releases a connection handle and frees all memory allocated for the handle.

For more information, see SQLFreeConnect in the ODBC Programmer’s Reference.

SQLFreeEnv (Visual FoxPro ODBC Driver)

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.</td>
</tr>
</tbody>
</table>

Support: Full

ODBC API Conformance: Core Level

Closes the Visual FoxPro ODBC Driver and releases all memory associated with the driver.

For more information, see SQLFreeEnv in the ODBC Programmer’s Reference.

SQLFreeStmt (Visual FoxPro ODBC Driver)

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.</td>
</tr>
</tbody>
</table>

Support: Full

ODBC API Conformance: Core Level

Stops processing associated with a specific hstmt, closes any open cursors associated with the hstmt, discards pending results, and optionally frees all resources associated with the statement handle.

For more information, see SQLFreeStmt in the ODBC Programmer’s Reference.

SQLGetConnectOption (Visual FoxPro ODBC Driver)

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.</td>
</tr>
</tbody>
</table>

Support: Partial

ODBC API Conformance: Level 1

Returns the current setting of a connection option. This function is partially supported: The driver supports all values for the fOption argument but does not support some of vParam values for the fOption argument SQL_TXN_ISOLATION.

The following table describes only those arguments with behavior specific to the Visual FoxPro ODBC Driver implementation of SQLGetConnectOption.
### SQL_AUTOCOMMIT

If you choose SQL_AUTOCOMMIT_OFF, your application must explicitly commit or roll back transactions with SQLTransact; the Visual FoxPro ODBC Driver does not automatically commit a transactable statement upon completion. The driver does begin a transaction if the statement is transactable.

### SQL_CURRENT_QUALIFIER

Can be a fully qualified database (.dbc file) name or fully qualified path to a directory containing zero or more tables (.dbf files).

### SQL_LOGINTIMEOUT

Returns "Driver Not Capable" error.

### SQL_CURSORS

Returns "Driver Not Capable" error.

### SQL_PACKET_SIZE

Returns "Driver Not Capable" error.

### SQL_TXN_ISOLATION

The driver allows only SQL_TXN_READ_COMMITTED. The following vParams are not supported:

- SQL_TXN_READ_UNCOMMITTED
- SQL_TXN_REPEATABLE_READ
- SQL_TXN_SERIALIZABLE

For more information, see SQLGetConnectOption in the ODBC Programmer's Reference.

© 2016 Microsoft

### SQLGetCursorName (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Returns the name of the cursor associated with the given hstmt. SQLGetCursorName is included in the Visual FoxPro ODBC Driver API because it is a part of Core Level API functionality; it cannot be used with other API functions because the driver does not support positioned updates.

For more information, see SQLGetCursorName in the ODBC Programmer's Reference.

© 2016 Microsoft

### SQLGetData (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 1

Retrieves the value of a single field in the current record of the given result set.

For more information, see SQLGetData in the ODBC Programmer's Reference.

© 2016 Microsoft

### SQLGetFunctions (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.
Support: Full

ODBC API Conformance: Level 1

Returns TRUE for all supported functions.

The Visual FoxPro ODBC Driver supports all ODBC API Core and Level 1 functions. The following table indicates whether the driver supports a specific Level 2 function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_API_SQLBROWSECONNECT</td>
<td>No</td>
</tr>
<tr>
<td>SQL_API_SQLCOLUMNPRIVILEGES</td>
<td>No</td>
</tr>
<tr>
<td>SQL_API_SQLDATASOURCES</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL_API_SQLDESCRIBEPARAM</td>
<td>No</td>
</tr>
<tr>
<td>SQL_API_SQLDRIVERS</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL_API_SQLEXTENDEDFETCH</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL_API_SQLFOREIGNKEYS</td>
<td>No</td>
</tr>
<tr>
<td>SQL_API_SQLMORERESULTS</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL_API_SQLNATIVESQL</td>
<td>No</td>
</tr>
<tr>
<td>SQL_API_SQLNUMPARAMS</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL_API_SQLPARAMOPTIONS</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL_API_SQLPRIMARYKEYS</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL_API_SQLPROCEDURECOLUMNS</td>
<td>No</td>
</tr>
<tr>
<td>SQL_API_SQLPROCEDURES</td>
<td>No</td>
</tr>
<tr>
<td>SQL_API_SQLSETPOS</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL_API_SQLSCROLLOPTIONS</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL_API_SQLTABLEPRIVILEGES</td>
<td>No</td>
</tr>
</tbody>
</table>

For more information, see SQLGetFunctions in the ODBC Programmer’s Reference.

© 2016 Microsoft

**SQLGetInfo (Visual FoxPro ODBC Driver)**

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 1

Returns general information about the Visual FoxPro ODBC Driver and data source associated with a connection handle, hdbc. The following list shows the value returned by the Visual FoxPro ODBC Driver for each fInfoType argument and comments regarding the returned values.

For more information, see SQLGetInfo in the ODBC Programmer’s Reference.

A

SQL_ACCESSIBLE_PROCEDURES returns 'N'.
SQL_ACCESSIBLE_TABLES returns 'Y'.
SQL_ACTIVE_CONNECTIONS returns 0.
SQL_ACTIVE_STATEMENTS returns 0.
SQL_ALTER_TABLE returns either SQL_AT_ADD_COLUMN or SQL_AT_DROP_COLUMN.

B

SQL_BOOKMARK_PERSISTENCE returns SQL_BP_SCROLL.

C

SQL_COLUMN_ALIAS returns 'Y'.
SQL_CONCAT_NULL_BEHAVIOR returns SQL_CB_NULL.
SQL_CONVERT_BIGINT returns 0. The Visual FoxPro ODBC Driver does not support BigInt.
SQL_CONVERT_BINARY returns 0.
SQL_CONVERT_BIT returns 0.
SQL_CONVERT_CHAR returns 0.
SQL_CONVERT_DATE returns 0.
SQL_CONVERT_DECIMAL returns 0.
SQL_CONVERT_DOUBLE returns 0.
SQL_CONVERT_FLOAT returns 0.
SQL_CONVERT_INTEGER returns 0.
SQL_CONVERT_LONGVARBINARY returns 0.
SQL_CONVERT_LONGVARCHAR returns 0.
SQL_CONVERT_NUMERIC returns 0.
SQL_CONVERT_REAL returns 0.
SQL_CONVERT_SMALLINT returns 0.
SQL_CONVERT_TIME returns 0.
SQL_CONVERT_TIMESTAMP returns 0.
SQL_CONVERT_TINYINT returns 0.
SQL_CONVERT_VARBINARY returns 0.
SQL_CONVERT_VARCHAR returns 0.
SQL_CONVERT_FUNCTIONS returns 0.
SQL_CORRELATION_NAME returns SQL_CN_ANY.
SQL_CURSOR_COMMIT_BEHAVIOR returns SQL_CB_PRESERVE.
SQL_CURSOR_ROLLBACK_BEHAVIOR returns SQL_CB_PRESERVE.

D

SQL_DATA_SOURCE_NAME returns the value passed as DSN to SQLConnect, or SQLDriverConnect; returns an empty string if no DSN is specified.
SQL_DATA_SOURCE_READ_ONLY returns 'N'.
SQL_DATABASE_NAME returns a full UNC path to the current database if the data source is a database. If the data source connects to a directory of tables, the function returns the path to the directory.
SQL_DBMS_NAME returns "Visual FoxPro".
SQL_DBMS_VER returns "03.00.0000".
SQL_DEFAULT_TXN_ISOLATION returns SQL_TXN_READ_COMMITTED. Dirty reads are not possible, but nonrepeatable reads and phantoms are possible.
SQL_DRIVER_HDBC is implemented by the Driver Manager.
SQL_DRIVER_HENV is implemented by the Driver Manager.
SQL_DRIVER_HLIB is implemented by the Driver Manager.
SQL_DRIVER_HSTMT is implemented by the Driver Manager.

SQL_DRIVER_NAME returns "vfpodbc.dll".

SQL_DRIVER_ODBC_VER returns "02.50" (SQL_SPEC_MAJOR, SQL_SPEC_MINOR).

SQL_DRIVER_VER returns "01.00.0000".

E

SQL_EXPRESSIONS_IN_ORDERBY returns 'N'.

F

SQL_FETCH_DIRECTION returns:

- SQL_FD_FETCH_NEXT
- SQL_FD_FETCH_FIRST
- SQL_FD_FETCH_LAST
- SQL_FD_FETCH_PRIOR
- SQL_FD_FETCH_ABSOLUTE
- SQL_FD_FETCH_RELATIVE
- SQL_FD_FETCH_BOOKMARK.

SQL_FILE_USAGE returns SQL_FILE_QUALIFIER both for database (.dbc file) and for free table (.dbf file) data sources.

G-H

SQL_GETDATA_EXTENSIONS returns:

- SQL_GD_ANY_COLUMN
- SQL_GD_ANY_BLOCK
- SQL_GD_ANY_BOUND
- SQL_GD_ANY_ORDER

SQL_GROUP_BY returns SQL_GB_NO_RELATION.

I-J

SQL_IDENTIFIER_CASE returns SQL_IC_MIXED.

SQL_IDENTIFIER_QUOTE_CHAR returns `.`.

K

SQL_KEYWORDS returns "".

L

SQL_LIKE_ESCAPE_CLAUSE returns 'N'.

SQL_LOCK_TYPES returns SQL_LCK_NO_CHANGE.

M

SQL_MAX_BINARY_LITERAL_LEN returns 0.
SQL_MAX_CHAR_LITERAL_LEN returns 254.
SQL_MAX_COLUMN_NAME_LEN returns 128.
SQL_MAX_COLUMNS_IN_GROUP_BY returns 16.
SQL_MAX_COLUMNS_IN_ORDER_BY returns 16.
SQL_MAX_COLUMNS_IN_INDEX returns 0.
SQL_MAX_COLUMNS_IN_SELECT returns 254.
SQL_MAX_COLUMNS_IN_TABLE returns 254.
SQL_MAX_CURSOR_NAME_LEN returns 254.
SQL_MAX_INDEX_SIZE returns 0.
SQL_MAX_OWNER_NAME_LEN returns 0.
SQL_MAX_PROCEDURE_NAME_LEN returns 0. The Visual FoxPro ODBC Driver does not allow direct access to Visual FoxPro stored procedures.
SQL_MAX_QUALIFIER_NAME_LEN returns the maximum operating system path length.
SQL_MAX_ROW_SIZE returns 254^2.
SQL_MAX_ROW_SIZE_INCLUDES_LONG returns 'N'.
SQL_MAX_STATEMENT_LEN returns 8192.
SQL_MAX_TABLE_NAME_LEN returns 128.
SQL_MAX_TABLES_IN_SELECT returns 16.
SQL_MAX_USER_NAME_LEN returns 0.
SQL_MULT_RESULT_SETS returns 'Y'.
SQL_MULTIPLE_ACTIVE_TXN returns 'Y'. Multiple connections can have several transactions open at once.

N
SQL_NEED_LONG_DATA_LEN returns 'N'.
SQL_NON_NULLABLE_COLUMNS returns SQL_NNC_NON_NULL.
SQL_NULL_COLLATION returns SQL_NC_LOW.

SQL_NUMERIC_FUNCTIONS returns all functions except SQL_FN_NUM_POWER, which is not supported by the Visual FoxPro ODBC Driver. The following functions are supported:

- SQL_FN_NUM_ABS
- SQL_FN_NUM_ACOS
- SQL_FN_NUM_ASIN
- SQL_FN_NUM_ATAN
- SQL_FN_NUM_ATAN2
- SQL_FN_NUM_CEILING
- SQL_FN_NUM_COS
- SQL_FN_NUM_COT
- SQL_FN_NUM_DEGREES
- SQL_FN_NUM_EXP
- SQL_FN_NUM_FLOOR
- SQL_FN_NUM_LOG
- SQL_FN_NUM_LOG10
- SQL_FN_NUM_MOD
- SQL_FN_NUM_PI
- SQL_FN_NUM_RADIANS
- SQL_FN_NUM_RAND
- SQL_FN_NUM_ROUND
- SQL_FN_NUM_SIGN
- SQL_FN_NUM_SIN
- SQL_FN_NUM_SQRT
- SQL_FN_NUM_TAN

O

SQL_ODBC_API_CONFORMANCE returns SQL_OAC_LEVEL1.
SQL_ODBC_SAG_CLI_CONFORMANCE returns SQL_OSCC_COMPLIANT.
SQL_ODBC_SQL_CONFORMANCE returns SQL_OSC_MINIMUM. Minimum SQL syntax is supported.
SQL_ODBC_SQL_OPT_IEF returns "N".
SQL_ODBC_VER is implemented by the Driver Manager.
SQL_ORDER_BY_COLUMNS_IN_SELECT returns "N".
SQL_OUTER_JOINS returns "N".
SQL_OWNER_TERM returns "". The Visual FoxPro ODBC Driver does not support owners for its objects.
SQL_OWNER_USAGE returns 0. The Visual FoxPro ODBC Driver does not support owners for its objects.

P

SQL_POS_OPERATIONS returns SQL_POS_POSITION.
SQL_POSITIONED_STATEMENTS returns 0.
SQL_PROCEDURE_TERM returns "".
SQL_PROCEDURES returns 'N'.

Q

SQL_QUALIFIER_LOCATION returns SQL_QL_START.
SQL_QUALIFIER_NAME_SEPARATOR returns '!' or '\'. The separator between database and table is '!' for data sources connected to databases, and '\' for data sources that are directories of free tables.
SQL_QUALIFIER_TERM returns "database" or "directory". The qualifier is "database" for data sources connected to databases, and "directory" for data sources that are directories of free tables.
SQL_QUALIFIER_USAGE does not support SQL_QU_PRIVILEGE_DEFINITION; it returns either SQL_QU_DML_STATEMENT or SQL_QU_TABLE_DEFINITION.
SQL_QUOTED_IDENTIFIER_CASE returns SQL_IC_MIXED.

R

SQL_ROW_UPDATES returns "N". The Visual FoxPro ODBC Driver supports only static and forward cursors.

S

SQL_SCROLL_CONCURRENCY returns SQL_SCCO_READ_ONLY.
SQL_SCROLL_OPTIONS returns either SQL_SO_STATIC or SQL_SO_READONLY.
SQL_SEARCH_PATTERN_ESCAPE returns "\".
SQL_SERVER_NAME returns "".
SQL_SPECIAL_CHARACTERS returns "~@#$%^".
SQL_STATIC_SENSITIVITY returns 0. The Visual FoxPro ODBC Driver does not support positional updates.
SQL_STRING_FUNCTIONS does not support SQL_FN_STR_INSERT, SQL_FN_STR_LOCATE, SQL_FN_STR_LOCATE_2, or SQL_FN_STR_SOUNDEX.
It returns:

- `SQL_FN_STR_ASCII`
- `SQL_FN_STR_CHAR`
- `SQL_FN_STR_CONCAT`
- `SQL_FN_STR_DIFFERENCE`
- `SQL_FN_STR_LCASE`
- `SQL_FN_STR_LEFT`
- `SQL_FN_STR_LENGTH`
- `SQL_FN_STR_LTRIM`
- `SQL_FN_STR_REPEAT`
- `SQL_FN_STR_REPLACE`
- `SQL_FN_STR_RIGHT`
- `SQL_FN_STR_RTRIM`
- `SQL_FN_STR_SUBSTRING`
- `SQL_FN_STR_UCASE`
- `SQL_FN_STR_SPACE`.

`SQL_SUBQUERIES` returns:

- `SQL_SQ_CORRELATED_SUBQUERIES`
- `SQL_SQ_COMPARISON`
- `SQL_SQ_EXISTS`
- `SQL_SQ_IN`
- `SQL_SQ_QUANTIFIED`.

`SQL_SYSTEM_FUNCTIONS` returns:

- `SQL_FN_SYS_DBNAME`
- `SQL_FN_SYS_IFNULL`

but not:

- `SQL_FN_SYS_USERNAME`

`T`

`SQL_TABLE_TERM` returns "table".

`SQL_TIMEDATE_ADD_INTERVALS` returns:

- `SQL_FN_TSI_SECOND`
- `SQL_FN_TSI_MINUTE`
- `SQL_FN_TSI_HOUR`
- `SQL_FN_TSI_DAY`
- `SQL_FN_TSI_MONTH`
- `SQL_FN_TSI_YEAR`

but not:

- `SQL_FN_TSI_FRAC_SECOND`
- `SQL_FN_TSI_WEEK`
- SQL_FN_TSI_QUARTER

SQL_TIMEDATE_DIFF_INTERVALS returns:
- SQL_FN_TSI_SECOND
- SQL_FN_TSI_MINUTE
- SQL_FN_TSI_HOUR
- SQL_FN_TSI_DAY
- SQL_FN_TSI_MONTH
- SQL_FN_TSI_YEAR

SQL_TIMEDATE_FUNCTIONS does not support SQL_FN_TD_QUARTER, SQL_FN_TD_TIMESTAMPADD, SQL_FN_TD_DAYOFYEAR, or SQL_FN_TD_WEEK.
It returns:
- SQL_FN_TD_CURDATE
- SQL_FN_TD_CURTIME
- SQL_FN_TD_DAYNAME
- SQL_FN_TD_DAYOFMONTH
- SQL_FN_TD_DAYOFWEEK
- SQL_FN_TD_HOUR
- SQL_FN_TD_MINUTE
- SQL_FN_TD_MONTH
- SQL_FN_TD_MONTHNAME
- SQL_FN_TD_NOW
- SQL_FN_TD_SECOND
- SQL_FN_TD_TIMESTAMPDIFF
- SQL_FN_TD_YEAR.

SQL_TXN_CAPABLE returns SQL_TC_DML.
SQL_TXN_ISOLATION_OPTION returns SQL_TXN_READ_COMMITTED.

U-Z

SQL_UNION returns either SQL_U_UNION or SQL_U_UNION_ALL.
SQL_USER_NAME returns <blank>.

© 2016 Microsoft

SQLGetStmtOption (Visual FoxPro ODBC Driver)

Note

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full
ODBC API Conformance: Level One

Returns the current setting of a statement option.

<table>
<thead>
<tr>
<th>FOption</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Visual FoxPro ODBC Driver has no translation DLLs.

For more information, see SQLGetStmtOption in the ODBC Programmer's Reference.

© 2016 Microsoft

### SQLGetTypeInfo (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 1

Returns information about the data types supported by a data source. The driver returns the information in an SQL result set. The following table lists ODBC data types and the corresponding Visual FoxPro data type.

<table>
<thead>
<tr>
<th>ODBC type</th>
<th>Visual FoxPro type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_BIGINT</td>
<td>Not supported. There is no 64-bit Visual FoxPro type.</td>
</tr>
<tr>
<td>SQL_BIT</td>
<td>Logical</td>
</tr>
<tr>
<td>SQL_CHAR</td>
<td>Character</td>
</tr>
<tr>
<td>SQL_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>SQL_DECIMAL</td>
<td>Numeric</td>
</tr>
<tr>
<td>SQL_DOUBLE</td>
<td>Double</td>
</tr>
<tr>
<td>SQL_FLOAT</td>
<td>Double</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>Integer</td>
</tr>
<tr>
<td>SQL_LONGVARBINARY</td>
<td>Memo (Binary)</td>
</tr>
<tr>
<td>SQL_LONGVARCHAR</td>
<td>Memo</td>
</tr>
<tr>
<td>SQL_NUMERIC</td>
<td>Numeric*, Currency, Float</td>
</tr>
<tr>
<td>SQL_REAL</td>
<td>Double</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>Integer</td>
</tr>
<tr>
<td>SQL_TIME</td>
<td>Not supported. There is no Visual FoxPro time type.</td>
</tr>
<tr>
<td>SQL_TIMESTAMP</td>
<td>DateTime</td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td>Integer</td>
</tr>
<tr>
<td>SQL_VARBINARY</td>
<td>Memo (Binary)*, General</td>
</tr>
<tr>
<td>SQL_VARCHAR</td>
<td>Character</td>
</tr>
</tbody>
</table>

*Default type

For more information about Visual FoxPro data types, see CREATE TABLE. For more information about this function, see SQLGetTypeInfo in the ODBC Programmer's Reference.
SQLMoreResults (Visual FoxPro ODBC Driver)

**Note**
This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 2

Determines whether more results are pending on a statement handle, hstmt, containing SELECT, UPDATE, INSERT, or DELETE statements and if so, initializes processing for those results.

For more information, see SQLMoreResults in the ODBC Programmer's Reference.

© 2016 Microsoft

SQLNumParams (Visual FoxPro ODBC Driver)

**Note**
This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 2

Returns the number of parameters in an SQL statement. The number of parameters should equal the number of question marks in the SQL statement passed to SQLPrepare.

For more information about SQL grammar, see Supported ODBC SQL Grammar. For more information about this function, see SQLNumParams in the ODBC Programmer's Reference.

© 2016 Microsoft

SQLNumResultCols (Visual FoxPro ODBC Driver)

**Note**
This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Returns the number of columns in a result set cursor.

For more information, see SQLNumResultCols in the ODBC Programmer's Reference.

© 2016 Microsoft

SQLParamData (Visual FoxPro ODBC Driver)

**Note**
This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 1

Used in conjunction with SQLPutData to specify parameter data at statement execution time.

For more information, see SQLParamData in the ODBC Programmer's Reference.
SQLParamOptions (Visual FoxPro ODBC Driver)

**Note**
This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full
ODBC API Conformance: Level 1

Allows an application to specify multiple values for the set of parameters assigned by `SQLBindParameter`. The ability to specify multiple values for a set of parameters is useful for bulk inserts and other work that requires the data source to process the same SQL statement multiple times with various parameter values. For example, an application can specify three sets of values for the set of parameters associated with an `INSERT` statement and then execute the `INSERT` statement once to perform the three insert operations.

For more information, see `SQLParamOptions` in the ODBC Programmer’s Reference.

SQLPrepare (Visual FoxPro ODBC Driver)

**Note**
This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full
ODBC API Conformance: Core Level

Prepares an SQL statement by planning how to optimize and execute the statement. The SQL statement is compiled for execution by `SQLExecDirect`.

If your table, view, or field names contain spaces, enclose the names in back quote (‘) marks. For example, if your database contains a table named My Table and the field My Field, enclose each element of the identifier as follows:

```sql
SELECT * FROM 'My Table'. 'My Field'
```

For more information, see `SQLPrepare` in the ODBC Programmer’s Reference.

SQLPrimaryKeys (Visual FoxPro ODBC Driver)

**Note**
This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full
ODBC API Conformance: Level 2

Returns the column names that comprise the primary key for a table. The Visual FoxPro ODBC Driver implementation of `SQLPrimaryKeys` behaves as follows:

- Ignores the `szTableOwner` and `cbTableOwner` arguments.
- Works only for data sources that are databases. The driver returns the error "Driver does not support this function" if the data source is a directory of free tables.

For more information, see `SQLPrimaryKeys` in the ODBC Programmer's Reference.

SQLPutData (Visual FoxPro ODBC Driver)
SQL Row Count (Visual FoxPro ODBC Driver)

**Note**
This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full
ODBC API Conformance: Core Level
Returns the number of rows affected by the last `UPDATE`, `INSERT`, or `DELETE` statement.
For more information, see `SQLRowCount` in the ODBC Programmer's Reference.

© 2016 Microsoft

SQLSetConnectOption (Visual FoxPro ODBC Driver)

**Note**
This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Partial
ODBC API Conformance: Level 1
Sets options that govern aspects of connections. This function is partially supported: The driver supports all values for the `fOption` argument but does not support some of `vParam` values for the `fOption` argument SQL_TXN_ISOLATION.

The following table describes only those arguments with behavior specific to the Visual FoxPro ODBC Driver implementation of `SQLSetConnectOption`.

<table>
<thead>
<tr>
<th><code>fOption</code></th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_AUTOCOMMIT</td>
<td>If you choose SQL_AUTOCOMMIT_OFF, your application must explicitly commit or roll back transactions with <code>SQLTransact</code>; the Visual FoxPro ODBC Driver does not automatically commit a transactable statement upon completion. The driver does begin a transaction if the statement is transactable.</td>
</tr>
<tr>
<td>SQL_CURRENT_QUALIFIER</td>
<td>Can be a fully qualified database name or fully qualified path to a directory containing zero or more free tables.</td>
</tr>
<tr>
<td>SQL_LOGINTIMEOUT</td>
<td>Returns &quot;Driver not capable&quot; error.</td>
</tr>
<tr>
<td>SQL_CURSORS</td>
<td>Returns &quot;Driver not capable&quot; error.</td>
</tr>
<tr>
<td>SQL_PACKET_SIZE</td>
<td>Returns &quot;Driver not capable&quot; error.</td>
</tr>
<tr>
<td>SQL_TXN_ISOLATION</td>
<td>The driver allows only SQL_TXN_READ_COMMITTED. The following vParams are not supported: SQL_TXN_READ_UNCOMMITTED SQL_TXN_REAPEATABE_READ SQL_TXN_SERIALIZABLE</td>
</tr>
</tbody>
</table>
SQLSetCursorName (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Associates a cursor name with an active statement handle, hstmt. SQLSetCursorName is included in the Visual FoxPro ODBC Driver API because it is a part of Core Level ODBC API functionality; it cannot be used with other API functions because the driver does not support positioned updates.

For more information, see SQLSetCursorName in the ODBC Programmer’s Reference.

SQLSetPos (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 2

Sets the cursor position in a rowset. You can use SQLSetPos with SQLGetData to retrieve rows from unbound columns after positioning the cursor to a specific row in the rowset.

For more information, see SQLSetPos in the ODBC Programmer’s Reference.

SQLSetScrollOptions (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Partial

ODBC API Conformance: Level 2

Sets options that control the behavior of cursors associated with a statement handle, hstmt.

The Visual FoxPro ODBC Driver supports only SQL_CONCUR_READ_ONLY; it does not support the fConcurrency value SQL_CONCUR_ROWVER. The driver converts SQL_KEYSET_SIZE, SQL_CURSOR_DYNAMIC, and SQL_CURSOR_KEYSET_DRIVEN to SQL_SCROLL_STATIC with warning ODBC_01S02.

For more information, see SQLSetScrollOptions in the ODBC Programmer’s Reference.

SQLSetStmtOption (Visual FoxPro ODBC Driver)

**Note**

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 1
Sets options related to a statement handle, hstmt.

<table>
<thead>
<tr>
<th>fOption</th>
<th>Allowed values</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ASYNC_ENABLE</td>
<td>SQL_ASYNC_ENABLE_OFF</td>
<td>If you attempt to set this option, the driver returns the error: “Driver not capable”. Visual FoxPro does not support asynchronous execution.</td>
</tr>
<tr>
<td>SQL_BIND_TYPE</td>
<td>SQL_BIND_BY_COLUMN or a 32-bit value denoting the length of the structure or an instance of a buffer into which result columns will be bound.</td>
<td>The driver doesn’t allow SQL_CONCUR_ROWVER, because Visual FoxPro does not have row versioning based on timestamps.</td>
</tr>
<tr>
<td>SQL_CONCURRENCY</td>
<td>SQL_CONCUR_READ_ONLY, SQL_CONCUR_LOCK, SQL_CONCUR_VALUES</td>
<td>The driver does not allow SQL_CONCUR_ROWVER, because Visual FoxPro does not have row versioning based on timestamps.</td>
</tr>
<tr>
<td>SQL_CURSOR_TYPE</td>
<td>SQL_CURSOR_FORWARD_ONLY, SQL_CURSOR_STATIC</td>
<td>The driver does not allow SQL_CURSOR_KEYSET_DRIVEN or SQL_CURSOR_DYNAMIC; see SQLSetScrollOptions for more information.</td>
</tr>
<tr>
<td>SQL_KEYSET_SIZE</td>
<td>Error: “Driver not capable”</td>
<td>Visual FoxPro does not support the keyset cursor model.</td>
</tr>
<tr>
<td>SQL_MAX_LENGTH</td>
<td>0</td>
<td>If you attempt to set this option value, the driver returns the error “Driver not capable”.</td>
</tr>
<tr>
<td>SQL_MAX_ROWS</td>
<td>0</td>
<td>If you attempt to set this option value, the driver returns the error “Driver not capable”.</td>
</tr>
<tr>
<td>SQL_NOSCAN</td>
<td>SQL_NOSCAN_OFF</td>
<td></td>
</tr>
<tr>
<td>SQL_QUERY_TIMEOUT</td>
<td>0</td>
<td>If you attempt to set this option value, the driver returns the error “Driver not capable”.</td>
</tr>
<tr>
<td>SQL_RETRIEVE_DATA</td>
<td>SQL_RD_ON, SQL_RD_OFF</td>
<td></td>
</tr>
<tr>
<td>SQL_ROWSET_SIZE</td>
<td>1 to 4,294,967,296</td>
<td></td>
</tr>
<tr>
<td>SQL_SIMULATE_CURSOR</td>
<td>Error: “Driver not capable”</td>
<td></td>
</tr>
<tr>
<td>SQL_USE_BOOKMARKS</td>
<td>SQL_UB_OFF, SQL_UB_ON</td>
<td></td>
</tr>
</tbody>
</table>

For more information, see SQLSetStmtOption in the ODBC Programmer’s Reference.

© 2016 Microsoft

SQLSpecialColumns (Visual FoxPro ODBC Driver)

Note

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 1

Retrieves the optimal set of columns that uniquely identifies a row in the table.

The Visual FoxPro ODBC Driver returns the columns that make up the primary key on the FoxPro table. (See SQLPrimaryKeys.) If called with fColType set to SQL_ROWVER, no columns are returned. SQLSpecialColumns works only for data sources that are databases.

For more information, see SQLSpecialColumns in the ODBC Programmer’s Reference.

© 2016 Microsoft

SQLStatistics (Visual FoxPro ODBC Driver)
Note

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 1

Retrieves a list of statistics about a single table and the indexes, or tag names, associated with the table. The driver returns the information as a result set.

For more information, see SQLStatistics in the ODBC Programmer’s Reference.

© 2016 Microsoft

SQLTables (Visual FoxPro ODBC Driver)

Note

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Level 1

Returns the list of table names specified by the parameter in the SQLTables statement. If no parameter is specified, returns the table names stored in the current data source. The driver returns the information as a result set.

Enumeration type calls will not receive a result set entry for remote views or local parameterized views. However, a call to SQLTables with a unique table name specifier will find a match for such a view if present with that name; this allows the API to be used to check for name conflicts prior to creation of a new table.

Note

The Visual FoxPro ODBC driver differentiates between database tables and free tables, even when both types of tables are stored in the same directory on your system. If your data source is a directory of free tables, the Visual FoxPro ODBC Driver does not catalog or return the names of any tables that are associated with a database.

For more information, see SQLTables in the ODBC Programmer’s Reference.

© 2016 Microsoft

SQLTransact (Visual FoxPro ODBC Driver)

Note

This topic contains Visual FoxPro ODBC Driver-specific information. For general information about this function, see the appropriate topic under ODBC API Reference.

Support: Full

ODBC API Conformance: Core Level

Requests a commit or rollback operation for all active operations on all statement handles (hstmts) associated with a connection or for all connections associated with the environment handle, henv. SQLTransact works only for data sources that are databases.

If a commit fails when in manual mode, the transaction remains active; you can choose to roll back the transaction or retry the commit operation. If a commit operation fails when in automatic transaction mode, the transaction is rolled back automatically; the transaction cannot be inactive.

For more information, see SQLTransact in the ODBC Programmer’s Reference.

© 2016 Microsoft

Visual FoxPro Language Reference

This section contains the following topics.

- ALTER TABLE - SQL Command
- CREATE TABLE - SQL Command
- DELETE - SQL Command
ALTER TABLE – SQL Command

Programmatically modifies the structure of a table.

Syntax

```
ALTER TABLE TableName1
    ADD | ALTER [COLUMN] FieldName1
         FieldType ([nFieldWidth [, nPrecision]])
         [NULL | NOT NULL]
         [CHECK lExpression1 [ERROR cMessageText1]]
         [DEFAULT eExpression1]
         [PRIMARY KEY | UNIQUE]
         [REFERENCES TableName2 [TAG TagName1]]
         [NOCPTRANS]
    - Or -
    ALTER TABLE TableName1
    ALTER [COLUMN] FieldName2
         [NULL | NOT NULL]
         [SET DEFAULT eExpression2]
         [SET CHECK lExpression2 [ERROR cMessageText2]]
         [DROP DEFAULT]
         [DROP CHECK]
    - Or -
    ALTER TABLE TableName1
    [DROP [COLUMN] FieldName3]
    [DROP CHECK]
    [ADD PRIMARY KEY eExpression3 [TAG TagName2]]
    [DROP PRIMARY KEY]
    [ADD UNIQUE eExpression4 [TAG TagName3]]
    [DROP UNIQUE TAG TagName4]
    [ADD FOREIGN KEY [TAG TagName5] TAG TagName4]
    [DROP FOREIGN KEY TAG TagName6 [SAVE]]
    [RENAME COLUMN FieldName4 TO FieldName5]
    [NOVALIDATE]
```
TableName1
Specifies the name of the table whose structure is modified.

ADD [COLUMN] FieldName1
Specifies the name of the field to add.

ALTER [COLUMN] FieldName1
Specifies the name of an existing field to modify.

FieldType [ (nFieldWidth [, nPrecision])]
Specifies the field type, field width, and field precision (number of decimal places) for a new or modified field.

FieldType is a single letter that indicates the field’s data type. Some field data types require that you specify nFieldWidth or nPrecision or both.

nFieldWidth and nPrecision are ignored for D, G, I, L, M, P, T, and Y types. By default, nPrecision is zero (no decimal places) if nPrecision is not included for the B, F, or N types.

NULL | NOT NULL
Allows or prevents null values in the field.

If you omit NULL and NOT NULL, the current setting of SET NULL determines whether null values are allowed in the field. However, if you omit NULL and NOT NULL and include the PRIMARY KEY or UNIQUE clause, the current setting of SET NULL is ignored and the field is NOT NULL by default.

CHECK lExpression1
Specifies a validation rule for the field. lExpression1 must evaluate to a logical expression and can be a user-defined function or a stored procedure.

Whenever a blank record is appended, the validation rule is checked. An error is generated if the validation rule does not allow for a blank field value in an appended record.

ERROR cMessageType1
Specifies the error message displayed when the field validation rule generates an error. The message is displayed only when data is changed within a Browse or Edit window.

DEFAULT eExpression1
Specifies a default value for an existing field. The data type of eExpression1 must be the same as the data type for the field.

PRIMARY KEY
Creates a primary index tag. The index tag has the same name as the field.

UNIQUE
Creates a candidate index tag with the same name as the field.

Note
Candidate indexes (created by including the UNIQUE option, provided for ANSI compatibility in ALTER TABLE or CREATE TABLE) differ from indexes created by using the UNIQUE option in the INDEX command. An index created by using UNIQUE in the INDEX command allows duplicate index keys; candidate indexes do not allow duplicate index keys.

Null values and duplicate records are not permitted in a field that is used for a primary or candidate index.

If you are creating a new field by using ADD COLUMN, Visual FoxPro will not generate an error if you create a primary or candidate index for a field that supports null values. However, Visual FoxPro will generate an error if you try to enter a null or duplicate value into a field that is used for a primary or candidate index.

If you are modifying an existing field and the primary or candidate index expression consists of fields in the table, Visual FoxPro checks the fields to see whether they contain null values or duplicate records. If they do, Visual FoxPro generates an error and the table is not altered.

REFERENCES TableName2 TAG TagName1
Specifies the parent table to which a persistent relationship is established. TAG TagName1 specifies the parent table's index tag on which the relationship is based. Index tag names can contain up to 10 characters.

NOCPTRANS
Prevents translation to a different code page for character and memo fields. If the table is converted to another code page, the fields for which NOCPTRANS has been specified are not translated. NOCPTRANS can be specified only for character and memo fields.

The following example creates a table named mytable that contains two character fields and two memo fields. The second character field, char2, and the second memo field, memo2, include NOCPTRANS to prevent translation.

CREATE TABLE mytable (char1 C(10), char2 C(10) NOCPTRANS,;
        memo1 M, memo2 M NOCPTRANS)

ALTER [COLUMN] FieldName2
Specifies the name of an existing field to modify.

SET DEFAULT eExpression2
Specifies a new default value for an existing field. The data type of eExpression2 must be the same as the data type for the field.

SET CHECK lExpression2
Specifies a new validation rule for an existing field. lExpression2 must evaluate to a logical expression and may be a user-defined function or a stored procedure.

ERROR cMessageType2
Specifies the error message displayed when the field validation rule generates an error. The message is displayed only when data is changed within a Browse or Edit window.
DROP DEFAULT
Removes the default value for an existing field.

DROP CHECK
Removes the validation rule for an existing field.

DROP [COLUMN] FieldName3
Specifies a field to remove from the table. Removing a field from the table also removes the field's default value setting and field validation rule.

If index key or trigger expressions reference the field, the expressions become invalid when the field is removed. In this case, an error is not generated when the field is removed but the invalid index key or trigger expressions will generate errors at run time.

SET CHECK eExpression3
Specifies the table validation rule. eExpression3 must evaluate to a logical expression and may be a user-defined function or a stored procedure.

ERROR cMessageText3
Specifies the error message displayed when the table validation rule generates an error. The message is displayed only when data is changed within a Browse or Edit window.

DROP CHECK
Removes the table's validation rule.

ADD PRIMARY KEY eExpression3[TAG TagName2]
Adds a primary index to the table. eExpression3 specifies the primary index key expression, and TagName2 specifies the name of the primary index tag. Index tag names can contain up to 10 characters. If TagTagName2 is omitted and eExpression3 is a single field, the primary index tag has the same name as the field specified in eExpression3.

DROP PRIMARY KEY
Removes the primary index and its index tag. Because a table can have only one primary key, it is not necessary to specify the name of the primary key. Removing the primary index also deletes any persistent relations based on the primary key.

ADD UNIQUE eExpression4[Tag TagName3]
Adds a candidate index to the table. eExpression4 specifies the candidate index key expression, and TagName3 specifies the name of the candidate index tag. Index tag names can contain up to 10 characters. If you omit TagTagName3 and if eExpression4 is a single field, the candidate index tag has the same name as the field specified in eExpression4.

DROP UNIQUE TAG TagName4
Removes the candidate index and its index tag. Because a table can have multiple candidate keys, you must specify the name of the candidate index tag.

ADD FOREIGN KEY [ eExpression5][TAG TagName4]
Adds a foreign (nonprimary) index to the table. eExpression5 specifies the foreign index key expression, and TagName4 specifies the name of the foreign index tag. Index tag names can contain up to 10 characters.

REFERENCES TableName2[Tag TagName5]
Specifies the parent table to which a persistent relationship is established. Include TagTagName5 to establish a relation based on an existing index tag for the parent table. Index tag names can contain up to 10 characters. If you omit TagTagName5, the relationship is established using the parent table's primary index tag.

DROP FOREIGN KEY TAG TagName6[SAVE]
Deletes a foreign key whose index tag is TagName6. If you omit SAVE, the index tag is deleted from the structural index. Include SAVE to prevent deletion of the index tag from the structural index.

RENAME COLUMN FieldName4TO FieldName5
Allows you to change the name of a field in the table. FieldName4 specifies the name of the field that is renamed. FieldName5 specifies the new name of the field.

<table>
<thead>
<tr>
<th>Caution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exercise care when renaming table fields because index expressions, field and table validation rules, commands, and functions may reference the original field names.</td>
</tr>
</tbody>
</table>

NOVALIDATE
Specifies that Visual FoxPro allows changes to be made to the structure of the table; these changes might violate the integrity of the data in the table. By default, Visual FoxPro prevents ALTER TABLE from making changes that violate the integrity of the data in the table. Include NOVALIDATE to override this default behavior.

Remarks

ALTER TABLE can be used to modify the structure of a table that has not been added to a database. However, Visual FoxPro generates an error if you include the DEFAULT, FOREIGN KEY, PRIMARY KEY, REFERENCES, or SET clauses when modifying a free table.

ALTER TABLE may rebuild the table by creating a new table header and appending records to the table header. For example, changing a field's type or width might cause the table to be rebuilt.

After a table is rebuilt, field validation rules are executed for any fields whose type or width is changed. If you change the type or width of any field in the table, the table rule is executed.

If you modify field or table validation rules for a table that has records, Visual FoxPro tests the new field or table validation rules against the existing data and issues a warning on the first occurrence of a field or table validation rule or of a trigger violation.

If the table you modify is in a database, ALTER TABLE - SQL requires exclusive use of the database. To open a database for exclusive use, include...
CREATE TABLE – SQL Command

Creates a table having the specified fields.

The Visual FoxPro ODBC Driver supports the native Visual FoxPro language syntax for this command. For driver-specific information, see Driver Remarks.

Syntax

```
CREATE TABLE [DBF] TableName1 [NAME LongTableName] [FREE]
   (FieldName1 FieldType [(nFieldWidth [, nPrecision])]
   [NULL | NOT NULL]
   [CHECK lExpression1 [ERROR cMessageText1]]
   [DEFAULT eExpression1]
   [PRIMARY KEY | UNIQUE]
   [REFERENCES TableName2 [TAG TagName1]]
   [NOCPTRANS]
   [, FieldName2 ...]
   [, PRIMARY KEY eExpression2 TAG TagName2
    | UNIQUE eExpression3 TAG TagName3]
   [, FOREIGN KEY eExpression4 TAG TagName4 [NODUP]
    REFERENCES TableName3 [TAG TagName5]]
   [, CHECK lExpression2 [ERROR cMessageText2]]
   [, DEFAULT eExpression1]
   [, PRIMARY KEY eExpression2 [TAG TagName2]]
   [, UNIQUE eExpression3 [TAG TagName3]]
   [, FOREIGN KEY eExpression4 [TAG TagName4] [NODUP]
    REFERENCES TableName3 [TAG TagName5]]
   [, CHECK lExpression2 [ERROR cMessageText2]]
   [, FROM ARRAY ArrayName]
```

Arguments

- **TableName1**: Specifies the name of the table to create. The TABLE and DBF options are identical.
- **NAME LongTableName**
  Specifies a long name for the table. A long table name can be specified only when a database is open, because long table names are stored in databases.
  Long names can contain up to 128 characters and can be used in place of short file names in the database.
- **FREE**: Specifies that the table will not be added to an open database. FREE isn’t required if a database isn’t open.
- **FieldName1 FieldType [(nFieldWidth [, nPrecision])]**
  Specifies the field name, field type, field width, and field precision (number of decimal places), respectively.
  
  - **FieldType**: a single letter indicating the field’s data type. Some field data types require that you specify nFieldWidth or nPrecision or both.
  
  - **nFieldWidth** and **nPrecision** are ignored for D, G, I, L, M, P, T, and Y types. nPrecision defaults to zero (no decimal places) if nPrecision isn’t included for the B, F, or N types.
- **NULL**
  Allows null values in the field.
- **NOT NULL**
  Prevents null values in the field.
  
  If you omit NULL and NOT NULL, the current setting of SET NULL determines whether null values are allowed in the field. However, if you omit NULL and NOT NULL and include the PRIMARY KEY or UNIQUE clause, the current setting of SET NULL is ignored and the field defaults to NOT NULL.
- **CHECK lExpression1**
  Specifies a validation rule for the field. lExpression1 can be a user-defined function. Whenever a blank record is appended, the validation rule is checked. An error is generated if the validation rule doesn’t allow for a blank field value in an appended record.
- **ERROR cMessageText1**
  Specifies the error message Visual FoxPro displays when the field rule generates an error. The message is displayed only when data is changed within a Browse window or an Edit window.
DEFAULT eExpression1
Specifies a default value for the field. The data type of eExpression1 must be the same as the field's data type.

PRIMARY KEY
Creates a primary index for the field. The primary index tag has the same name as the field.

UNIQUE
Creates a candidate index for the field. The candidate index tag has the same name as the field.

Note
Candidate indexes (created by including the UNIQUE option in CREATE TABLE or ALTER TABLE - SQL) are not the same as indexes created with the UNIQUE option in the INDEX command. An index created with the UNIQUE option in the INDEX command allows duplicate index keys; candidate indexes do not allow duplicate index keys. See INDEX for additional information on its UNIQUE option.

NOCPRTRANS
Prevents translation to a different code page for character and memo fields. If the table is converted to another code page, the fields for which NOCPTRANS has been specified are not translated. NOCPTRANS can be specified only for character and memo fields.

The following example creates a table named mytable containing two character fields and two memo fields. The second character field, char2, and the second memo field, memo2, include NOCPTRANS to prevent translation.

```
CREATE TABLE mytable (char1 C(10), char2 C(10) NOCPTRANS, memo1 M, memo2 M NOCPTRANS)
```

PRIMARY KEY eExpression2 TAG TagName1
Specifies a primary index to create. eExpression2 specifies any field or combination of fields in the table. TagName1 specifies the name for the primary index tag that is created. Index tag names can contain up to 10 characters.

Because a table can have only one primary index, you cannot include this clause if you have already created a primary index for a field. Visual FoxPro generates an error if you include more than one PRIMARY KEY clause in CREATE TABLE.

UNIQUE eExpression3 TAG TagName2
Creates a candidate index. eExpression3 specifies any field or combination of fields in the table. However, if you have created a primary index with one of the PRIMARY KEY options, you cannot include the field that was specified for the primary index. TagName2 specifies a tag name for the candidate index tag that is created. Index tag names can contain up to 10 characters.

A table can have multiple candidate indexes.

FOREIGN KEY eExpression4 TAG TagName5 [NODUP]
Creates a foreign (nonprimary) index and establishes a relationship to a parent table. eExpression4 specifies the foreign index key expression, and TagName5 specifies the name of the foreign index key tag that is created. Index tag names can contain up to 10 characters. Include NODUP to create a candidate foreign index.

You can create multiple foreign indexes for the table, but the foreign index expressions must specify different fields in the table.

REFERENCES TableName6 [TAG TagName6]
Specifies the parent table to which a persistent relationship is established. Include TagName6 to establish a relation based on an index tag for the parent table. Index tag names can contain up to 10 characters. By default, if you omit TagName6, the relationship is established using the parent table's primary index key.

CHECK eExpression7 [ERROR cMessageText2]
Specifies the table validation rule. ERROR cMessageText2 specifies the error message Visual FoxPro displays when the table validation rule is executed. The message is displayed only when data is changed within a Browse window or Edit window.

FROM ARRAY ArrayName
Specifies the name of an existing array whose contents are the name, type, precision, and scale for each field in the table. The contents of the array can be defined with the AFIELDS( ) function.

Remarks
The new table is opened in the lowest available work area and can be accessed by its alias. The new table is opened exclusively, regardless of the current setting of SET EXCLUSIVE.

If a database is open and you don't include the FREE clause, the new table is added to the database. You cannot create a new table with the same name as a table in the database.
If a database is open, CREATE TABLE - SQL requires exclusive use of the database. To open a database for exclusive use, include EXCLUSIVE in OPEN DATABASE.

If a database isn’t open when you create the new table, including the NAME, CHECK, DEFAULT, FOREIGN KEY, PRIMARY KEY, or REFERENCES clauses generates an error.

**Note**

CREATE TABLE syntax uses commas to separate certain CREATE TABLE options. Also, the NULL, NOT NULL, CHECK, DEFAULT, PRIMARY KEY, and UNIQUE clause must be placed within the parentheses containing the column definitions.

**Driver Remarks**

When your application sends the ODBC SQL statement CREATE TABLE to the data source, the Visual FoxPro ODBC Driver translates the command into the Visual FoxPro CREATE TABLE command using the syntax shown in the following table.

<table>
<thead>
<tr>
<th>ODBC syntax</th>
<th>Visual FoxPro syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE base-table-name</td>
<td>CREATE TABLE TableName1 [NAME LongTableName]</td>
</tr>
<tr>
<td>(column-identifier data type</td>
<td>(FieldName1 FieldType</td>
</tr>
<tr>
<td>[NOT NULL]</td>
<td>[nFieldWidth [, nPrecision]])</td>
</tr>
<tr>
<td>[,column-identifier data type</td>
<td>[NOT NULL])</td>
</tr>
<tr>
<td>[NOT NULL] ...)</td>
<td></td>
</tr>
</tbody>
</table>

When you create a table using the driver, the driver closes the table immediately after creation to allow access to the table by other users. This differs from Visual FoxPro, which leaves the table open exclusively upon creation. However, if a stored procedure on your data source containing a CREATE TABLE statement executes, the table is left open.

If the data source is a database (.dbc file), the Visual FoxPro ODBC Driver creates a table named LongTableName with the same name as the base-table-name.

**Using Data Definition Language (DDL)**

You cannot include DDL in the following places:

- In a batch SQL statement that requires a transaction
- In manual-commit mode, after a statement that required a transaction, unless your application first calls `SQLTransact`.

For example, if you want to create a temporary table, you should create the table before you begin the statement requiring a transaction. If you include the CREATE TABLE statement in a batch SQL statement that requires a transaction, the driver returns an error message.

**See Also**

- **ALTER TABLE - SQL Command**
- **Supported Data Types (Visual FoxPro ODBC Driver)**
- **INSERT - SQL Command**
- **SELECT - SQL Command**

**DELETE – SQL Command**

Marks records for deletion.

The Visual FoxPro ODBC Driver supports the native Visual FoxPro language syntax for this command. For driver-specific information, see the Remarks.

**Syntax**

```
DELETE FROM [DatabaseName!]TableName
    [WHERE FilterCondition1 [AND | OR FilterCondition2 ...]]
```
Arguments

FROM [DatabaseName!] TableName
Specifies the table in which records are marked for deletion.

DatabaseName! specifies the name of a database that contains the table if the containing database is not the database specified with the data source. You must include the name of a database that contains the table if the database is not the database specified with the data source. Include the exclamation point (!) delimiter after the database name and before the table name.

WHERE FilterCondition1[AND | OR FilterCondition2...] 
Specifies that Visual FoxPro mark only certain records for deletion. FilterCondition specifies the criteria that records must meet to be marked for deletion. You can include as many filter conditions as you want, connecting them with the AND or OR operator. You can also use the NOT operator to reverse the value of a logical expression, or you can use EMPTY( ) to check for an empty field.

Remarks

If SET DELETED is set to ON, records marked for deletion are ignored by all commands that include a scope.

DELETE - SQL uses record locking when marking multiple records for deletion in tables opened for shared access. This reduces record contention in multiuser situations but can decrease performance. For maximum performance, open the table for exclusive use.

Driver Remarks

When your application sends the ODBC SQL statement DELETE to the data source, the Visual FoxPro ODBC Driver converts the command into the Visual FoxPro DELETE command without translation.

See Also

SET DELETED Command

© 2016 Microsoft

DELETE TAG Command

Removes a tag or tags from a compound index (.cdx) file.

Syntax

```
DELETE TAG TagName1 [OF CDXFileName1] 
[, TagName2 [OF CDXFileName2]] ... 
0r
DELETE TAG ALL [OF CDXFileName]
```

Arguments

```
TagName1OF CDXFileName1[, TagName2[OF CDXFileName2]] ... 
```

Specifies a tag to remove from a compound index file. You can delete multiple tags with one DELETE TAG by including a list of tag names separated by commas. If two or more tags with the same name exist in the open index files, you can remove a tag from a specific index file by including OF CDXFileName.

```
ALL [OF CDXFileName]
```

Removes every tag from a compound index file. If the current table has a structural compound index file, all tags are removed from the index file, the index file is deleted from the disk, and the flag in the table's header indicating the presence of an associated structural compound index file is removed. Use ALL with OF CDXFileName to remove all tags from an open compound index file other than the structural compound index file.

Remarks
Compound index files, created with INDEX, contain tags corresponding to index entries. DELETE TAG is used to remove a tag or tags from open compound index files. You can delete only tags from compound index files open in the current work area. If you remove all the tags from a compound index file, the file is deleted from the disk.

Visual FoxPro looks first for a tag in the structural compound index file (if one is open). If the tag isn't in the structural compound index file, Visual FoxPro then looks for the tag in the other open compound index files.

See Also
INDEX Command

DROP TABLE Command

Removes a table from the database specified with the data source and deletes it from disk.

The Visual FoxPro ODBC Driver supports the native Visual FoxPro language syntax for this command. For driver-specific information, see the Remarks.

Syntax

```
DROP TABLE TableName | FileName | ?
```

Settings

**TableName**
Specifies the table to remove from the database specified with the data source and to delete from disk.

**FileName**
Specifies a free table to delete from disk.

? Displays the Remove dialog from which you can choose a table to remove from the database specified with the data source and to delete from disk.

Remarks

When DROP TABLE is issued, all primary indexes, default values, and validation rules associated with the table are also removed. DROP TABLE also affects other tables in the database specified with the data source if those tables have rules or relations associated with the table being removed. The rules and relations are no longer valid when the table is removed from the database.

Driver Remarks

When your application sends the ODBC SQL statement DROP TABLE to the data source, the Visual FoxPro ODBC Driver converts the command into the Visual FoxPro DROP TABLE command using the syntax shown in the following table.

<table>
<thead>
<tr>
<th>ODBC syntax</th>
<th>Data source</th>
<th>Visual FoxPro syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP TABLE base-table-name</td>
<td>Database (.dbc file)</td>
<td>REMOVE TABLE TableName DELETE</td>
</tr>
<tr>
<td></td>
<td>Directory of free tables (.dbf files)</td>
<td>ERASE dbfName</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ERASE cdxName</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ERASE fptName</td>
</tr>
</tbody>
</table>

INDEX Command
Syntax

INDEX ON eExpression TO IDXFileName | TAG TagName [OF CDXFileName]
[FOR lExpression]
[COMPACT]
[ASCENDING | DESCENDING]
[UNIQUE | CANDIDATE]
[ADDITIVE]

Arguments

eExpression
States an index expression that can include the name of a field or fields from the current table. An index key based on the index expression is created in the index file for each record in the table. Visual FoxPro uses these keys to display and access records in the table.

Note
Although not recommended, eExpression can also be a memory variable, an array element, or a field or field expression from a table in another work area. Memo fields cannot be used alone in index file expressions; they must be combined with other character expressions. If you access an index that contains a variable or field that no longer exists or cannot be located, Visual FoxPro generates an error message.

If you attempt to build an index with a key that varies in length, the key will be padded with spaces. Variable-length index keys aren’t supported in Visual FoxPro.

It is possible to create an index key with zero length. For example, a zero length index key is created when the index expression is a substring of an empty memo field. A zero length index key generates an error message. When Visual FoxPro creates an index, it evaluates fields in the first record in the table. If a field is empty, it might be necessary to enter some temporary data in the field in the first record to prevent a 0-length index key.

TO IDXFileName
Creates an .idx index file. The index file is given the default extension .idx.

TAG TagName [OF CDXFileName]
Creates a compound index file. A compound index file is a single index file that consists of any number of separate tags (index entries). Each tag is identified by its unique tag name. Tag names must begin with a letter or an underscore and can consist of any combination of up to 10 letters, digits, or underscores. The number of tags in a compound index file is limited only by available memory and disk space.

Multiple-entry compound index files are always compact. It isn’t necessary to include COMPACT when creating a compound index file. Names of compound index files are given a .cdx extension.

Two types of compound index files can be created: structural and nonstructural.

Structural Compound Index Files You can create a structural compound index file with TAG TagName by excluding the optional OF CDXFileName clause. A structural compound index file always has the same base name as the table and is automatically opened when the table is opened.

Nonstructural Compound Index Files You can create a nonstructural compound index file by including OF CDXFileName after TagName. Unlike a structural compound index file, a nonstructural compound index file must be explicitly opened with the INDEX clause in USE.

If a compound index file has already been created and opened, issuing INDEX with TAG TagName adds a tag to the compound index file.

FOR lExpression
Specifies a condition whereby only records that satisfy the filter expression lExpression are available for display and access; index keys are created in the index file for just those records matching the filter expression.

Visual FoxPro Rushmore technology optimizes an INDEX ... FOR lExpression command if lExpression is an optimizable expression. For best performance, use an optimizable expression in the FOR clause.

COMPACT
Creates a compact .idx file.

ASCENDING
Specifies an ascending order for the .cdx file. By default, .cdx tags are created in ascending order. (You can include ASCENDING as a reminder of the index file’s order.) A table can be indexed in reverse order by including DESCENDING.

DESCENDING
Specifies a descending order for the .cdx file. You can’t include DESCENDING when creating .idx index files.

UNIQUE
Specifies that only the first record encountered with a particular index key value is included in an .idx file or a .cdx tag. UNIQUE can be used to prevent the display of or access to duplicate records. All records added with duplicate index keys are excluded from the index file. Using the UNIQUE option of INDEX is identical to executing SET UNIQUE ON before issuing INDEX or REINDEX.

When a UNIQUE index or index tag is active and a duplicate record is changed in a manner that changes its index key, the index or index tag is updated. However, the next duplicate record with the original index key cannot be accessed or displayed until you reindex the file using REINDEX.

CANDIDATE
Creates a candidate structural index tag. The CANDIDATE keyword can be included only when creating a structural index tag; otherwise, Visual FoxPro generates an error message.

A candidate index tag prevents duplicate values in the field or combination of fields specified in the index expression eExpression. The term candidate refers to the type of index; because candidate indexes prevent duplicate values, they qualify as a "candidate" to be a primary index.

Visual FoxPro generates an error if you create a candidate index tag for a field or combination of fields that already contains duplicate values.

**ADDITIVE**

Keeps open any previously opened index files. If you omit the ADDITIVE clause when you create an index file or files for a table with INDEX, any previously opened index files (except the structural compound index) are closed.

**Remarks**

Records in a table that has an index file are displayed and accessed in the order specified by the index expression. The physical order of the records in the table isn’t changed by an index file.

**Index Types**

Visual FoxPro lets you create two types of index files:

- Compound .cdx index files containing multiple index entries called tags
- .idx index files containing one index entry

You can also create a structural compound index file, which is automatically opened with the table.

**Note**

Because structural compound index files are automatically opened when the table is opened, they are the preferred index type.

Include COMPACT to create compact .idx index files. Compound index files are always compact.

**Index Order and Updating**

Only one index file (the master index file) or tag (the master tag) controls the order in which the table is displayed or accessed. Certain commands (SEEK, for example) use the master index file or tag to search for records. However, all open .idx and .cdx index files are updated as changes are made to the table.

**User-Defined Functions**

Although an index expression can contain a user-defined function, you should not use user-defined functions in an index expression. User-defined functions in an index expression increase the time it takes to create or update the index. Also, index updates might not occur when a user-defined function is used for an index expression.

If you use a user-defined function in an index expression, Visual FoxPro must be able to locate the user-defined function. When Visual FoxPro creates an index, the index expression is saved in the index file but only a reference to the user-defined function is included in the index expression.

**See Also**

- ALTER TABLE - SQL Command
- DELETE TAG Command
- SET COLLATE Command
- SET UNIQUE Command

© 2016 Microsoft

**INSERT – SQL Command**

Appends a record to the end of a table that contains the specified field values.

The Visual FoxPro ODBC Driver supports the native Visual FoxPro language syntax for this command. For driver-specific information, see the Remarks.

**Syntax**
INSERT INTO dbf_name 
VALUES (eExpression1 [, eExpression2, ...])

Arguments

INSERT INTO dbf_name
Specifies the name of the table to which the new record is appended. dbf_name can include a path and can be a name expression.

If the table you specify isn't open, it is opened exclusively in a new work area and the new record is appended to the table. The new work area isn't selected; the current work area remains selected.

If the table you specify is open, INSERT appends the new record to the table. If the table is open in a work area other than the current work area, it isn't selected after the record is appended; the current work area remains selected.

VALUES ([fname1 [, fname2, ...]])
Specifies in the new record the names of the fields into which the values are inserted.

VALUES ( eExpression1 [, eExpression2 [, ...]])
Specifies the field values inserted into the new record. If you omit the field names, you must specify the field values in the order defined by the table structure.

Remarks

The new record contains the data listed in the VALUES clause.

Driver Remarks

When your application sends the ODBC SQL statement INSERT to the data source, the Visual FoxPro ODBC Driver converts the command into the Visual FoxPro INSERT command without translation.

See Also

CREATE TABLE - SQL Command
SELECT - SQL Command

© 2016 Microsoft

SELECT – SQL Command

Retrieves data from one or more tables.

The Visual FoxPro ODBC Driver supports the native Visual FoxPro language syntax for this command. For driver-specific information, see Driver Remarks.

Syntax

```
SELECT [ALL | DISTINCT]  
    [Alias.] Select_Item [AS Column_Name]  
    [, [Alias.] Select_Item [AS Column_Name] ...]  
FROM [DatabaseName!]Table [Local_Alias]  
    [, [DatabaseName!]Table [Local_Alias] ...]  
    [WHERE JoinCondition [AND JoinCondition ...]]  
    [AND | OR FilterCondition [AND | OR FilterCondition ...]])  
    [GROUP BY GroupColumn [, GroupColumn ...]]  
    [HAVING FilterCondition]  
    [UNION [ALL] SELECTCommand]  
    [ORDER BY Order_Item [ASC | DESC] [, Order_Item [ASC | DESC] ...]]
```

Arguments


A subquery, referred to in the following arguments, is a SELECT within a SELECT and must be enclosed in parentheses. You can have up to two subqueries at the same level (not nested) in the WHERE clause. (See that section of the arguments.) Subqueries can contain multiple join conditions.

**Note**

The SELECT clause specifies the fields, constants, and expressions that are displayed in the query results.

By default, ALL displays all the rows in the query results.

DISTINCT excludes duplicates of any rows from the query results.

**Note**

You can use DISTINCT only once per SELECT clause.

Aliasing qualifying matching item names. Each item you specify with Select_Item generates one column of the query results. If two or more items have the same name, include the table alias and a period before the item name to prevent columns from being duplicated.

Select_Item specifies an item to be included in the query results. An item can be one of the following:

- The name of a field from a table in the FROM clause.
- A constant specifying that the same constant value is to appear in every row of the query results.
- An expression that can be the name of a user-defined function.

**User-Defined Functions with SELECT**

Although using user-defined functions in the SELECT clause has obvious benefits, you should also consider the following restrictions:

- The speed of operations performed with SELECT might be limited by the speed at which such user-defined functions are executed. High-volume manipulations involving user-defined functions might be better accomplished by using API and user-defined functions written in C or assembly language.
- The only reliable way to pass values to user-defined functions invoked from SELECT is by the argument list passed to the function when it is invoked.
- Even if you experiment and discover a supposedly forbidden manipulation that works correctly in a certain version of FoxPro, there is no guarantee it will continue to work in later versions.

Apart from these restrictions, user-defined functions are acceptable in the SELECT clause. However, remember that using SELECT might slow performance.

The following field functions are available for use with a select item that is a field or an expression involving a field:

- AVG(Select_Item)—Averages a column of numeric data.
- COUNT(Select_Item)—Counts the number of select items in a column. COUNT(*) counts the number of rows in the query output.
- MIN(Select_Item)—Determines the smallest value of Select_Item in a column.
- MAX(Select_Item)—Determines the largest value of Select_Item in a column.
- SUM(Select_Item)—Totals a column of numeric data.

You cannot nest field functions.

**AS Column_Name**

Specifies the heading for a column in the query output. This is useful when Select_Item is an expression or contains a field function and you want to give the column a meaningful name. Column_Name can be an expression but cannot contain characters (for example, spaces) that are not permitted in table field names.

**FROM**

Lists the tables that contain the data that the query retrieves. If no table is open, Visual FoxPro displays the Open dialog box so that you can specify the file location. After it has been opened, the table remains open after the query is complete.

DatabaseName! specifies the name of a database other than the one specified with the data source. You must include the name of the database that contains the table if the database is not specified with the data source. Include the exclamation point (!) delimiter after the database name and before the table name.

Local_Alias specifies a temporary name for the table named in Table. If you specify a local alias, you must use the local alias instead of the table name throughout the SELECT statement. The local alias does not affect the Visual FoxPro environment.

**WHERE**

Tells Visual FoxPro to include only certain records in the query results. WHERE is required to retrieve data from multiple tables.

JoinCondition specifies fields that link the tables in the FROM clause. If you include more than one table in a query, you should specify a join condition for every table after the first.
Consider the following information when you create join conditions:

- If you include two tables in a query and do not specify a join condition, every record in the first table is joined to every record in the second table as long as the filter conditions are met. Such a query can produce lengthy results.

- Use caution when joining tables with empty fields because Visual FoxPro matches empty fields. For example, if you join on CUSTOMER.ZIP and INVOICE.ZIP and if CUSTOMER contains 100 empty zip codes and INVOICE contains 400 empty zip codes, the query output contains 40,000 extra records resulting from the empty fields. Use the `EMPTY( )` function to eliminate empty records from the query output.

- You must use the AND operator to connect multiple join conditions. Each join condition has the following form:

  FieldName1 Comparison FieldName2

  FieldName1 is the name of a field from one table, FieldName2 is the name of a field from another table, and Comparison is one of the operators described in the following table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>==</td>
<td>Exactly equal</td>
</tr>
<tr>
<td>LIKE</td>
<td>SQL LIKE</td>
</tr>
<tr>
<td>&lt;&gt;, !=, #</td>
<td>Not equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>More than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>More than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
</tbody>
</table>

When you use the `=` operator with strings, it acts differently, depending on the setting of SET ANSI. When SET ANSI is set to OFF, Visual FoxPro treats string comparisons in a manner familiar to Xbase users. When SET ANSI is set to ON, Visual FoxPro follows ANSI standards for string comparisons. See SET ANSI and SET EXACT for more information about how Visual FoxPro performs string comparisons.

FilterCondition specifies the criteria that records must meet to be included in the query results. You can include as many filter conditions in a query as you want, connecting them with the AND or OR operator. You can also use the NOT operator to reverse the value of a logical expression, or you can use `EMPTY( )` to check for an empty field. FilterCondition can take any of the forms in the following examples:

**Example 1**  
FieldName1 Comparison FieldName2  
customer.cust_id = orders.cust_id

**Example 2**  
FieldName Comparison Expression  
payments.amount >= 1000

**Example 3**  
FieldName Comparison ALL (Subquery)  
company < ALL ;  
(SELECT company FROM customer WHERE country = "USA")

When the filter condition includes ALL, the field must meet the comparison condition for all values generated by the subquery before its record is included in the query results.

**Example 4**  
FieldName Comparison ANY | SOME (Subquery)  
company < ANY ;  
(SELECT company FROM customer WHERE country = "USA")

When the filter condition includes ANY or SOME, the field must meet the comparison condition for at least one of the values generated by the subquery.

The following example checks to see whether the values in the field are within a specified range of values:

**Example 5**  
FieldName [NOT] BETWEEN Start_Range AND End_Range  
customer.postalcode BETWEEN 90000 AND 99999

The following example checks to see whether at least one row meets the criteria in the subquery. When the filter condition includes EXISTS, the filter condition evaluates to True (.T.) unless the subquery evaluates to the empty set.

**Example 6**  
[NOT] EXISTS (Subquery)
exists;
(SELECT * FROM orders WHERE customer.postalcode =
orders.postalcode)

**Example 7**  
**FieldName** [NOT] **IN** **Value_Set**

customer.postalcode NOT IN ("98052", "98072", "98034")

When the filter condition includes IN, the field must contain one of the values before its record is included in the query results.

**Example 8**  
**FieldName** [NOT] **IN** (Subquery)

customer.cust_id IN ;
(SELECT orders.cust_id FROM orders WHERE orders.city="Seattle")

Here the field must contain one of the values returned by the subquery before its record is included in the query results.

**Example 9**  
**FieldName** [NOT] **LIKE** cExpression

customer.country NOT LIKE "USA"

This filter condition searches for each field that matches cExpression. You can use the percent sign (%) and underscore (_ ) wildcard characters as part of cExpression. The underscore represents a single unknown character in the string.

**GROUP BY** GroupColumn [ , GroupColumn ... ]

Groups rows in the query based on values in one or more columns. GroupColumn can be one of the following:

- The name of a regular table field.
- A field that includes an SQL field function.
- A numeric expression that indicates the location of the column in the result table. (The leftmost column number is 1.)

**HAVING** FilterCondition

Specifies a filter condition that groups must meet to be included in the query results. HAVING should be used with GROUP BY and can include as many filter conditions as you want, connected by the AND or OR operator. You can also use NOT to reverse the value of a logical expression.

FilterCondition cannot contain a subquery.

A HAVING clause without a GROUP BY clause behaves like a WHERE clause. You can use local aliases and field functions in the HAVING clause. Use a WHERE clause for faster performance if your HAVING clause contains no field functions.

[UNION [ALL] SELECTCommand]

Combines the final results of one SELECT with the final results of another SELECT. By default, UNION checks the combined results and eliminates duplicate rows. Use parentheses to combine multiple UNION clauses.

ALL prevents UNION from eliminating duplicate rows from the combined results.

UNION clauses follow these rules:

- You cannot use UNION to combine subqueries.
- Both SELECT commands must have the same number of columns in their query output.
- Each column in the query results of one SELECT must have the same data type and width as the corresponding column in the other SELECT.
- Only the final SELECT can have an ORDER BY clause, which must refer to output columns by number. If an ORDER BY clause is included, it affects the complete result.

You can also use the UNION clause to simulate an outer join.

When you join two tables in a query, only records with matching values in the joining fields are included in the output. If a record in the parent table does not have a corresponding record in the child table, the record in the parent table is not included in the output. An outer join lets you include all the records in the parent table in the output, together with the matching records in the child table. To create an outer join in Visual FoxPro, you must use a nested SELECT command, as in the following example:

```sql
SELECT customer.company, orders.order_id, orders.emp_id ;
FROM customer, orders ;
WHERE customer.cust_id = orders.cust_id ;
UNION ;
SELECT customer.company, 0, 0 ;
FROM customer ;
WHERE customer.cust_id NOT IN ;
(SELECT orders.cust_id FROM orders)
```

**Note**
The section of the command before the UNION clause selects records from both tables that have matching values. The customer companies that do not have associated invoices are not included. The section of the command after the UNION clause selects records in the customer table that do not have matching records in the orders table.

Regarding the second section of the command, note the following:

- The SELECT statement within the parentheses is processed first. This statement creates a selection of all customer numbers in the orders table.
- The WHERE clause finds all customer numbers in the customer table that are not in the orders table. Because the first section of the command provided all companies that had a customer number in the orders table, all companies in the customer table are now included in the query results.
- Because the structures of tables included in a UNION must be identical, there are two placeholders in the second SELECT statement to represent `orders.order_id` and `orders.emp_id` from the first SELECT statement.

Note

The placeholders must be the same type as the fields that they represent. If the field is a date type, the placeholder should be `{ / / }`. If the field is a character field, the placeholder should be the empty string (```).

ORDER BY Order_Item [ASC | DESC ] [ , Order_Item [ASC | DESC ] ]

Sorts the query results based on the data in one or more columns. Each Order_Item must correspond to a column in the query results and can be one of the following:

- A field in a FROM table that is also a select item in the main SELECT clause (not in a subquery).
- A numeric expression that indicates the location of the column in the result table. (The leftmost column is number 1.)

ASC specifies an ascending order for query results, according to the order item or items, and is the default for ORDER BY. DESC specifies a descending order for query results.

Query results appear unordered if you do not specify an order with ORDER BY.

Remarks

SELECT is an SQL command that is built into Visual FoxPro like any other Visual FoxPro command. When you use SELECT to pose a query, Visual FoxPro interprets the query and retrieves the specified data from the tables. You can create a SELECT query from within either the Command Prompt window or a Visual FoxPro program (as with any other Visual FoxPro command).

Note

SELECT does not respect the current filter condition specified with SET FILTER.

Driver Remarks

When your application sends the ODBC SQL statement SELECT to the data source, the Visual FoxPro ODBC Driver converts the command into the Visual FoxPro SELECT command without translation unless the command contains an ODBC escape sequence. Items enclosed in an ODBC escape sequence are converted to Visual FoxPro syntax. For more information about using ODBC escape sequences, see Time and Date Functions and in the Microsoft ODBC Programmer’s Reference, see Escape Sequences in ODBC.

See Also

CREATE TABLE - SQL
INSERT - SQL
SET ANSI
SET EXACT

© 2016 Microsoft

SET ANSI Command

Determines how comparisons between strings of different lengths are made with the = operator in Visual FoxPro SQL commands.
Syntax

```
SET ANSI ON | OFF
```

Arguments

**ON**
(Default for the driver; the default for Visual FoxPro is OFF.) Pads the shorter string with the blanks needed to make it equal to the longer string's length. The two strings are then compared character for character for their entire lengths. Consider this comparison:

```
'Tommy' = 'Tom'
```

The result is False (.F.) if SET ANSI is on, because when padded, 'Tom' becomes 'Tom ' and the strings 'Tom ' and 'Tommy' don't match character for character.

The == operator uses this method for comparisons in Visual FoxPro SQL commands.

**OFF**
Specifies that the shorter string not be padded with blanks. The two strings are compared character for character until the end of the shorter string is reached. Consider this comparison:

```
'Tommy' = 'Tom'
```

The result is True (.T.) when SET ANSI is off, because the comparison stops after 'Tom'.

Remarks

SET ANSI determines whether the shorter of two strings is padded with blanks when an SQL string comparison is made. SET ANSI has no effect on the == operator; when you use the == operator, the shorter string is always padded with blanks for the comparison.

String Order

In SQL commands, the left-to-right order of the two strings in a comparison is irrelevantswitching a string from one side of the = or == operator to the other doesn't affect the result of the comparison.

See Also

- SELECT - SQL Command
- SET EXACT Command

© 2016 Microsoft

**SET BLOCKSIZE Command**

Specifies how disk space is allocated for the storage of memo fields.

Syntax

```
SET BLOCKSIZE TO nBytes
```

Arguments

*nBytes*
Specifies the block size in which disk space for memo fields is allocated. If \textit{nBytes} is 0, disk space is allocated in single bytes (blocks of 1 byte). If \textit{nBytes} is an integer between 1 and 32, disk space is allocated in blocks of \textit{nBytes} bytes multiplied by 512. If \textit{nBytes} is greater than 32, disk space is allocated in blocks of \textit{nBytes} bytes. If you specify a block size value greater than 32, you can save substantial disk space.

**Remarks**

The default value for \texttt{SET BLOCKSIZE} is 64. To reset the block size to a different value after the file has been created, set it to a new value and then use \texttt{COPY} to create a new table. The new table has the specified block size.

---

**SET COLLATE Command**

Specifies a collation sequence for character fields in subsequent indexing and sorting operations.

**Syntax**

```plaintext
SET COLLECT TO cSequenceName
```

**Arguments**

\textit{cSequenceName}

Specifies a collation sequence. The available collation sequence options are described in the following table.

<table>
<thead>
<tr>
<th>Options</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUTCH</td>
<td>Dutch</td>
</tr>
<tr>
<td>GENERAL</td>
<td>English, French, German, Modern Spanish, Portuguese, and other Western European languages</td>
</tr>
<tr>
<td>GERMAN</td>
<td>German phone book order (DIN)</td>
</tr>
<tr>
<td>ICELAND</td>
<td>Icelandic</td>
</tr>
<tr>
<td>MACHINE</td>
<td>Machine (the default collation sequence for earlier FoxPro versions)</td>
</tr>
<tr>
<td>NORDAN</td>
<td>Norwegian, Danish</td>
</tr>
<tr>
<td>SPANISH</td>
<td>Traditional Spanish</td>
</tr>
<tr>
<td>SWEFIN</td>
<td>Swedish, Finnish</td>
</tr>
<tr>
<td>UNIQWT</td>
<td>Unique Weight</td>
</tr>
</tbody>
</table>

**Note**

When you specify the SPANISH option, \textit{ch} is a single letter that sorts between \textit{c} and \textit{d}, and \textit{ll} sorts between \textit{l} and \textit{m}.

If you specify a collation sequence option as a literal character string, be sure to enclose the option in quotation marks:

```plaintext
SET COLLECT TO "SWEFIN"
```

\texttt{MACHINE} is the default collation sequence option and is the sequence Xbase users are familiar with. Characters are ordered as they appear in the current code page.

\texttt{GENERAL} may be preferable for U.S. and Western European users. Characters are ordered as they appear in the current code page. In FoxPro versions earlier than 2.5, indexes might have been created using the \texttt{UPPER( )} or \texttt{LOWER( )} functions to convert character fields to a consistent case. In FoxPro versions later than 2.5, you can instead specify the \texttt{GENERAL} collation sequence option and omit the \texttt{UPPER( )} conversion.

If you specify a collation sequence option other than \texttt{MACHINE} and if you create an .idx file, a compact .idx is always created.

Use \texttt{SET("COLLATE")} to return the current collation sequence.
You can specify a collating sequence for a data source by using the ODBC Visual FoxPro Setup Dialog Box or by using the Collate keyword in your connection string with SQLDriverConnect. This is identical to issuing the following command:

```
SET COLLATE TO cSequenceName
```

Remarks

SET COLLATE enables you to order tables containing accented characters for any of the supported languages. Changing the setting of SET COLLATE doesn't affect the collating sequence of previously opened indexes. Visual FoxPro automatically maintains existing indexes, providing the flexibility to create many different types of indexes, even for the same field.

For example, if an index is created with SET COLLATE set to GENERAL and the SET COLLATE setting is later changed to SPANISH, the index retains the GENERAL collation sequence.

See Also

ODBC Visual FoxPro Setup Dialog Box

SET DELETED Command

Specifies whether records marked for deletion are processed and whether they are available for use in other commands.

Syntax

```
SET DELETED ON | OFF
```

Arguments

ON

(Default for the driver; the default for Visual FoxPro is OFF.) Specifies that commands that operate on records (including records in related tables) using a scope ignore records marked for deletion.

OFF

Specifies that records marked for deletion can be accessed by commands that operate on records (including records in related tables) using a scope.

Remarks

Queries that use `DELETED( )` to test the status of records can be optimized using Visual FoxPro Rushmore technology if the table is indexed on `DELETED( )`.

Important

SET DELETED is ignored if the default scope for the command is the current record or if you include a scope of a single record. INDEX always ignores SET DELETED and indexes all records in the table.

See Also

DELETE - SQL Command

SET EXACT Command
Specifies the rules for comparing two strings of different lengths.

Syntax

```
SET EXACT ON | OFF
```

Arguments

ON

Specifies that expressions must match character for character to be equivalent. Any trailing blanks in the expressions are ignored for the comparison. For the comparison, the shorter of the two expressions is padded on the right with blanks to match the length of the longer expression.

OFF

(Default.) Specifies that, to be equivalent, expressions must match character for character until the end of the expression on the right side is reached.

Remarks

The SET EXACT setting has no effect if both strings are the same length.

String Comparisons

Visual FoxPro has two relational operators that test for equality.

The = operator performs a comparison between two values of the same type. This operator is suited for comparing character, numeric, date, and logical data.

However, when you compare character expressions with the = operator, the results might not be exactly what you expect. Character expressions are compared character for character from left to right until one of the expressions isn't equal to the other, until the end of the expression on the right side of the = operator is reached (SET EXACT OFF), or until the ends of both expressions are reached (SET EXACT ON).

The == operator can be used when an exact comparison of character data is needed. If two character expressions are compared with the == operator, the expressions on both sides of the == operator must contain exactly the same characters, including blanks, to be considered equal. The SET EXACT setting is ignored when character strings are compared using ==.

The following table shows how the choice of operator and the SET EXACT setting affect comparisons. (An underscore represents a blank space.)

<table>
<thead>
<tr>
<th>Comparison</th>
<th>= EXACT OFF</th>
<th>= EXACT ON</th>
<th>== EXACT ON or OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot; = &quot;abc&quot;</td>
<td>Match</td>
<td>Match</td>
<td>Match</td>
</tr>
<tr>
<td>&quot;ab&quot; = &quot;abc&quot;</td>
<td>No match</td>
<td>No match</td>
<td>No match</td>
</tr>
<tr>
<td>&quot;abc&quot; = &quot;ab&quot;</td>
<td>Match</td>
<td>No match</td>
<td>No match</td>
</tr>
<tr>
<td>&quot;abc&quot; = &quot;ab_&quot;</td>
<td>No match</td>
<td>No match</td>
<td>No match</td>
</tr>
<tr>
<td>&quot;ab&quot; = &quot;ab_&quot;</td>
<td>No match</td>
<td>Match</td>
<td>No match</td>
</tr>
<tr>
<td>&quot;ab_&quot; = &quot;ab&quot;</td>
<td>Match</td>
<td>Match</td>
<td>No match</td>
</tr>
<tr>
<td>&quot;&quot; = &quot;ab&quot;</td>
<td>No match</td>
<td>No match</td>
<td>No match</td>
</tr>
<tr>
<td>&quot;ab&quot; = &quot;&quot;</td>
<td>Match</td>
<td>No match</td>
<td>No match</td>
</tr>
<tr>
<td>&quot;_&quot; = &quot;&quot;</td>
<td>Match</td>
<td>Match</td>
<td>No match</td>
</tr>
<tr>
<td>&quot;&quot; = &quot;_&quot;</td>
<td>No match</td>
<td>Match</td>
<td>No match</td>
</tr>
<tr>
<td>TRIM(&quot;___&quot;) = &quot;&quot;</td>
<td>Match</td>
<td>Match</td>
<td>Match</td>
</tr>
<tr>
<td>&quot;&quot; = TRIM(&quot;___&quot;)</td>
<td>Match</td>
<td>Match</td>
<td>Match</td>
</tr>
</tbody>
</table>

See Also

SET EXACT ON | OFF
SET EXCLUSIVE Command

Specifies whether table files are opened for exclusive or shared use on a network.

Syntax

```
SET EXCLUSIVE ON | OFF
```

Arguments

ON

Limits accessibility of a table opened on a network to the user who opened it. The table isn't accessible to other users on the network. SET EXCLUSIVE ON also prevents all other users from having read-only access.

OFF

(Default for the driver; the defaults for Visual FoxPro are ON for the global data session and OFF for a private data session.) Allows a table opened on a network to be shared and modified by any user on the network.

Remarks

Changing the setting of SET EXCLUSIVE doesn't change the status of previously opened tables. For example, if a table is opened with SET EXCLUSIVE set to ON and SET EXCLUSIVE is later changed to OFF, the table retains its exclusive-use status.

See Also

ODBC Visual FoxPro Setup Dialog Box

SET NULL Command

Determines how null values are supported by the ALTER TABLE - SQL, CREATE TABLE - SQL, and INSERT - SQL commands.

Syntax

```
SET NULL ON | OFF
```

Arguments

ON

(Default for the driver; the default for Visual FoxPro is OFF.) Specifies that all columns in a table created with ALTER TABLE and CREATE TABLE will allow null values. You can override null value support for columns in the table by including the NOT NULL clause in the columns' definitions.

Also specifies that INSERT - SQL will insert null values into any columns not included in the INSERT - SQL VALUE clause. INSERT - SQL will insert null values only into columns that allow null values.

OFF

Specifies that all columns in a table created with ALTER TABLE and CREATE TABLE will not allow null values. You can designate null value support for columns in ALTER TABLE and CREATE TABLE by including the NULL clause in the columns' definitions.

Also specifies that INSERT - SQL will insert blank values into any columns not included in the INSERT - SQL VALUE clause.
Remarks

SET NULL affects only how null values are supported by ALTER TABLE, CREATE TABLE, and INSERT - SQL. Other commands are unaffected by SET NULL.

See Also

ALTER TABLE - SQL Command
CREATE TABLE - SQL Command
INSERT - SQL Command

SET PATH Command

Specifies a path for file searches. For driver-specific information, see the Remarks.

Syntax

```
SET PATH TO [Path]
```

Arguments

TO [Path]

Specifies the directories you want Visual FoxPro to search. Use commas or semicolons to separate the directories.

Remarks

SET PATH allows you to specify search paths for other Visual FoxPro programs that can be called within stored procedures. SET PATH will not change the path of the data source that you’ve specified for the connection.

Issue SET PATH TO without Path to restore the path to the default directory or folder.

Driver Remarks

If you issue SET PATH in a stored procedure, it will be ignored by the following functions and commands:

- Catalog functions such as SQLTables and SQLColumns will ignore the new path and continue to reference the path specified by the data source in SQLPrepare or SQLExecDirect.
- Commands such as SELECT, INSERT, UPDATE, DELETE, and CREATE TABLE will ignore the new path and continue to reference the path specified by the data source in SQLPrepare or SQLExecDirect.

If you issue SET PATH in a stored procedure and don’t subsequently set the path back to its original state, other connections to the database will use the new path (because SET PATH is not scoped to data sessions).

If you want to create, select, or update tables in a directory other than that specified by the data source, specify the full path of the file with your command.

See Also

ODBC Visual FoxPro Setup Dialog Box
SQLColumns (Visual FoxPro ODBC Driver)
SQLDriverConnect (Visual FoxPro ODBC Driver)
SQLTables (Visual FoxPro ODBC Driver)
SET REPROCESS Command

Specifies how many times or for how long to lock a file or record after an unsuccessful locking attempt.

Syntax

```
SET REPROCESS TO nAttempts [SECONDS] | TO AUTOMATIC
```

Arguments

**TO nAttempts[SECONDS]**

Specifies the number of times or number of seconds to try to lock a record or file after an initial unsuccessful attempt. The default value is 0; the maximum value is 32,000.

SECONDS specifies that Visual FoxPro attempts to lock a file or record for \( n \text{Attempts} \) seconds. It's available only when \( n \text{Attempts} \) is greater than zero.

For example, if \( n \text{Attempts} \) is 30, Visual FoxPro attempts to lock a record or file up to 30 times. If you also include SECONDS (SET REPROCESS TO 30 SECONDS), Visual FoxPro continuously attempts to lock a record or file for up to 30 seconds.

If an ON ERROR routine is in effect and if attempts by a command to lock the record or file are unsuccessful, the ON ERROR routine is executed. However, if a function attempts the lock, an ON ERROR routine isn't executed and the function returns False (.F.).

If an ON ERROR routine isn't in effect, a command attempts to lock the record or file, and the lock can't be placed, an error is generated. If a function attempts to place the lock, the alert isn't displayed and the function returns False (.F.).

If \( n \text{Attempts} \) is 0 (the default value) and you issue a command or function that attempts to lock a record or file, Visual FoxPro tries to lock the record or file indefinitely. If the record or file becomes available for locking while you wait, the lock is placed and the system message is cleared. If a function attempted to place the lock, the function returns True (.T.).

If an ON ERROR routine is in effect and a command is attempting to lock the record or file, the ON ERROR routine takes precedence over additional attempts to lock the record or file. The ON ERROR routine is immediately executed. Visual FoxPro does not attempt additional record or file locks and does not display the system message.

If \( n \text{Attempts} \) is 1, Visual FoxPro attempts to lock the record or file indefinitely and an ON ERROR routine isn't executed.

If a lock has been placed by another user on the record or file you are attempting to lock, you must wait until the user releases the lock.

**TO AUTOMATIC**

Specifies that Visual FoxPro attempts to lock the record or file indefinitely. (SET REPROCESS TO -2 is an equivalent command.)

Remarks

The first attempt to lock a record or file isn't always successful. Frequently, a record or file is locked by another user on the network. SET REPROCESS determines whether Visual FoxPro makes additional attempts to lock the record or file when the initial attempt is unsuccessful. You can specify either how many times additional attempts are made or for how long the attempts are made. An ON ERROR routine affects how unsuccessful lock attempts are handled.

© 2016 Microsoft

SET UNIQUE Command

Specifies whether records with duplicate index key values are maintained in an index file.

Syntax

```
SET UNIQUE ON | OFF
```

Arguments

ON
Specifies that any record with a duplicate index key value not be included in the index file. Only the first record with the original index key value is included in the index file.

OFF
(Default.) Specifies that records with duplicate index key values be included in the index file.

Remarks

An index file retains its SET UNIQUE setting when you issue REINDEX. For more information, see INDEX.

UPDATE – SQL Command

Updates records in a table with new values.

The Visual FoxPro ODBC Driver supports the native Visual FoxPro language syntax for this command. For driver-specific information, see Driver Remarks.

Syntax

```
UPDATE [DatabaseName1!] TableName1
SET Column_Name1 = eExpression1
 [, Column_Name2 = eExpression2 ...]
WHERE FilterCondition1 [AND | OR FilterCondition2 ...]
```

Arguments

UPDATE [ DatabaseName1!] TableName1
Specifies the table in which records are updated with new values.

DatabaseName1! specifies the name of a database other than the database specified with the data source containing the table. You must include the name of the database containing the table if the database is not the current one. Include the exclamation point (!) delimiter after the database name and before the table name.

SET Column_Name1 = eExpression1[, Column_Name2 = eExpression2]
Specifies the columns that are updated and their new values. If you omit the WHERE clause, every row in the column is updated with the same value.

WHERE FilterCondition1 [AND | OR FilterCondition2 ...]
Specifies the records that are updated with new values.

FilterCondition specifies the criteria that records must meet to be updated with new values. You can include as many filter conditions as you like, connecting them with the AND or OR operator. You can also use the NOT operator to reverse the value of a logical expression, or you can use EMPTY( ) to check for an empty field.

Remarks

UPDATE - SQL can update only records in a single table.

Unlike REPLACE, UPDATE - SQL uses record locking when updating multiple records in tables opened for shared access. This reduces record contention in multiuser situations but can reduce performance. For maximum performance, open the table for exclusive use or use FLOCK( ) to lock the table.

Driver Remarks

When your application sends the ODBC SQL statement UPDATE to the data source, the Visual FoxPro ODBC Driver converts the command into the Visual FoxPro UPDATE command without translation.

See Also

DELETE - SQL Command
INSERT - SQL Command
Visual FoxPro Field Data Types

The following table lists the values for the FieldType argument in ALTER TABLE and CREATE TABLE and indicates whether nFieldWidth and nPrecision arguments are required.

<table>
<thead>
<tr>
<th>FieldType</th>
<th>nFieldWidth</th>
<th>nPrecision</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>-</td>
<td>d</td>
<td>Double</td>
</tr>
<tr>
<td>C</td>
<td>N</td>
<td>-</td>
<td>Character field of width n</td>
</tr>
<tr>
<td>D</td>
<td>-</td>
<td>-</td>
<td>Date</td>
</tr>
<tr>
<td>F</td>
<td>N</td>
<td>d</td>
<td>Floating numeric field of width n with d decimal places</td>
</tr>
<tr>
<td>G</td>
<td>-</td>
<td>-</td>
<td>General</td>
</tr>
<tr>
<td>I</td>
<td>-</td>
<td>-</td>
<td>Integer</td>
</tr>
<tr>
<td>L</td>
<td>-</td>
<td>-</td>
<td>Logical</td>
</tr>
<tr>
<td>M</td>
<td>-</td>
<td>-</td>
<td>Memo</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>d</td>
<td>Numeric field of width n with d decimal places</td>
</tr>
<tr>
<td>T</td>
<td>-</td>
<td>-</td>
<td>DateTime</td>
</tr>
<tr>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>Currency</td>
</tr>
</tbody>
</table>

ODBC Test

Microsoft® ODBC Test is an ODBC-enabled application that you can use to test ODBC drivers and the ODBC Driver Manager. ODBC 3.51 includes both ANSI and Unicode-enabled versions of ODBC Test. The corresponding files are as follows:

- Odbcte32.exe and Gtrtst32.dll, for the ANSI version.
- Odbct32w.exe and Gtrts32w.dll, for the Unicode version.

To use ODBC Test, you must understand the ODBC API, the C language, and SQL. For more information about the ODBC API, see the ODBC Programmer’s Reference.

Help topics that were formerly included within this section of the documentation are now contained within the ODBC Test program. Open Odbcte32.exe or Odbct32w.exe, open the Help menu, and then click Help Topics.

Note that the 64-bit versions of these applications, meant for 64-bit Microsoft Windows operating systems, have the same names as the 32-bit versions, even though they are separate files. i.e. the name for the Unicode version of the 64-bit version of ODBC Test is odbct32w.exe.

ODBC Glossary

A

access plan
A plan generated by the database engine to execute an SQL statement. Equivalent to executable code compiled from a third-generation language such as C.

aggregate function
A function that generates a single value from a group of values, often used with GROUP BY and HAVING clauses. Aggregate functions include AVG, COUNT, MAX, MIN, and SUM. Also known as set functions. See also scalar function.

ANSI
American National Standards Institute. The ODBC API is based on the ANSI Call-Level Interface.
APD  
See application parameter descriptor (APD).

API  
Application Programming Interface. A set of routines that an application uses to request and carry out lower-level services. The ODBC API is composed of the ODBC functions.

application  
An executable program that calls functions in the ODBC API.

application parameter descriptor (APD)  
A descriptor that describes the dynamic parameters used in an SQL statement before any conversion specified by the application.

application row descriptor (ARD)  
A descriptor that represents the column metadata and data in the application's buffers, describing a row of data following any data conversion specified by the application.

ARD  
See application row descriptor (ARD).

auto-commit mode  
A transaction commit mode in which transactions are committed immediately after they are executed.

B  
behavioral change  
A change in certain functionality from ODBC 3.x behavior to ODBC 2.x behavior, or vice versa. Caused by changing the SQL_ATTR_ODBC_VERSION environment attribute.

Binary large object (BLOB)  
Any binary data over a certain number of bytes, such as 255. Typically much longer. Such data is generally sent to and retrieved from the data source in parts. Also known as long data.

binding  
As a verb, the act of associating a column in a result set or a parameter in an SQL statement with an application variable. As a noun, the association.

binding offset  
A value added to the data buffer addresses and length/indicator buffer addresses for all bound column or parameter data, producing new addresses.

block cursor  
A cursor capable of fetching more than one row of data at a time.

buffer  
A piece of application memory used to pass data between the application and driver. Buffers often come in pairs: a data buffer and a data length buffer.

byte  
Eight bits or one octet. See also octet.

C  
C data type  
The data type of a variable in a C program, in this case the application.

catalog  
The set of system tables in a database that describe the shape of the database. Also known as a schema or data dictionary.

catalog function  
An ODBC function used to retrieve information from the database's catalog.

CLI  
See API.

client/server  
A database access strategy in which one or more clients access data through a server. The clients usually implement the user interface while the server controls database access.

column  
The container for a single item of information in a row. Also known as field.

commit  
To make the changes in a transaction permanent.

concurrency  
The ability of more than one transaction to access the same data at the same time.

conformance level  
A discrete set of functionality supported by a driver or data source. ODBC defines API conformance levels and SQL conformance levels.

connection
A particular instance of a driver and data source.

**connection browsing**
Searching the network for data sources to connect to. Connection browsing might involve several steps. For example, the user might first browse the network for servers and then browse a particular server for a database.

**connection handle**
A handle to a data structure that contains information about a connection.

**current row**
The row currently pointed to by the cursor. Positioned operations act on the current row.

**cursor**
A piece of software that returns rows of data to the application. Probably named after the blinking cursor on a computer terminal; just as that cursor indicates the current position on the screen, a cursor on a result set indicates the current position in the result set.

---

**data buffer**
A buffer used to pass data. Often associated with a data buffer is a data length buffer.

**data dictionary**
See catalog.

**data length buffer**
A buffer used to pass the length of the value in a corresponding data buffer. The data length buffer is also used to store indicators, such as whether the data value is null-terminated.

**data source**
The data that the user wants to access and its associated operating system, DBMS, and network platform (if any).

**data type**
The type of a piece of data. ODBC defines C and SQL data types. See also type indicator.

**data-at-execution column**
A column for which data is sent after `SQLSetPos` is called. So named because the data is sent at execution time rather than being placed in a rowset buffer. Long data is generally sent in parts at execution time.

**data-at-execution parameter**
A parameter for which data is sent after `SQLExecute` or `SQLExecDirect` is called. So named because the data is sent when the SQL statement is executed rather than being placed in a parameter buffer. Long data is generally sent in parts at execution time.

**database**
A discrete collection of data in a DBMS. Also a DBMS.

**database engine**
The software in a DBMS that parses and executes SQL statements and accesses the physical data.

**DBMS**
Database Management System. A layer of software between the physical database and the user. The DBMS manages all access to the database.

**DBMS-based driver**
A driver that accesses physical data through a stand-alone database engine.

**DDL**
Data Definition Language. Those statements in SQL that define, as opposed to manipulate, data. For example, `CREATE TABLE`, `CREATE INDEX`, `GRANT`, and `REVOKE`.

**delimited identifier**
An identifier that is enclosed in identifier quote characters so that it can contain special characters or match keywords (also known as a quoted identifier).

**descriptor**
A data structure that holds information about either column data or dynamic parameters. The physical representation of the descriptor is not defined; applications gain direct access to a descriptor only by manipulating its fields by calling ODBC functions with the descriptor handle.

**desktop database**
A DBMS designed to run on a personal computer. Generally, these DBMSs do not provide a stand-alone database engine and must be accessed through a file-based driver. The engines in these drivers generally have reduced support for SQL and transactions. For example, dBASE, Paradox, Btrieve, or Microsoft® FoxPro®.

**diagnostic**
A record containing diagnostic information about the last function called that used a particular handle. Diagnostic records are associated with environment, connection, statement, and descriptor handles.

**DML**
Data Manipulation Language. Those statements in SQL that manipulate, as opposed to define, data. For example, `INSERT`, `UPDATE`, `DELETE`, and `SELECT`.

**driver**
A routine library that exposes the functions in the ODBC API. Drivers are specific to a single DBMS.

**Driver Manager**
A routine library that manages access to drivers for the application. The Driver Manager loads and unloads (or connects to and disconnects from) drivers and passes calls to ODBC functions to the correct driver.

**driver setup DLL**
A DLL that contains driver-specific installation and configuration functions.

**dynamic cursor**
A scrollable cursor capable of detecting rows updated, deleted, or inserted in the result set.

**dynamic SQL**
A type of embedded SQL in which SQL statements are created and compiled at run time. See also static SQL.

**embedded SQL**
SQL statements that are included directly in a program written in another language, such as COBOL or C. ODBC does not use embedded SQL. See also static SQL and dynamic SQL.

**environment**
A global context in which to access data; associated with the environment is any information that is global in nature, such as a list of all connections in that environment.

**environment handle**
A handle to a data structure that contains information about the environment.

**escape clause**
A clause in an SQL statement.

**execute**
To run an SQL statement.

**fat cursor**
See block cursor.

**fetch**
To retrieve one or more rows from a result set.

**field**
See column.

**file-based driver**
A driver that accesses physical data directly. In this case, the driver contains a database engine and acts as both driver and data source.

**file data source**
A data source for which connection information is stored in a .dsn file.

**foreign key**
A column or columns in a table that match the primary key in another table.

**forward-only cursor**
A cursor that can only move forward through the result set and generally fetches only one row at a time. Most relational databases support only forward-only cursors.

**handle**
A value that uniquely identifies something such as a file or data structure. Handles are meaningful only to the software that creates and uses them but are passed by other software to identify things. ODBC defines handles for environments, connections, statements, and descriptors.

**implementation parameter descriptor (IPD)**
A descriptor that describes the dynamic parameters used in an SQL statement after any conversion specified by the application.

**implementation row descriptor (IRD)**
A descriptor that describes a row of data before any conversion specified by the application.

**installer DLL**
A DLL that installs ODBC components and configures data sources.

**Integrity Enhancement Facility**
A subset of SQL designed to maintain the integrity of a database.

**interface conformance level**
The level of the ODBC 3.7 interface supported by a driver; can be Core, Level 1, or Level 2.

**interoperability**
The ability of one application to use the same code when accessing data in different DBMSs.

**IPD**
See Implementation Parameter Descriptor (IPD).

**IRD**
See Implementation Row Descriptor (IRD).

**ISO/IEC**

**J**

**join**
An operation in a relational database that links the rows in two or more tables by matching values in specified columns.

**K**

**key**
A column or columns whose values identify a row. See also foreign key and primary key.

**keyset**
A set of keys used by a mixed or keyset-driven cursor to refetch rows.

**keyset-driven cursor**
A scrollable cursor that detects updated and deleted rows by using a keyset.

**L**

**literal**
A character representation of an actual data value in an SQL statement.

**locking**
The process by which a DBMS restricts access to a row in a multiuser environment. The DBMS usually sets a bit on a row or the physical page containing a row that indicates the row or page is locked.

**long data**
Any binary or character data over a certain length, such as 255 bytes or characters. Typically much longer. Such data is generally sent to and retrieved from the data source in parts. Also known as BLOBs or CLOBs.

**M**

**machine data source**
A data source for which connection information is stored on the system (for example, the registry).

**manual-commit mode**
A transaction commit mode in which transactions must be explicitly committed by calling SQLTransact.

**metadata**
Data that describes a parameter in an SQL statement or a column in a result set. For example, the data type, byte length, and precision of a parameter.

**multiple-tier driver**
See DBMS-based driver.

**N**

**NULL value**
Having no explicitly assigned value. In particular, a NULL value is different from a zero or a blank.
octet
Eight bits or one byte. See also byte.

octet length
The length in octets of a buffer or the data it contains.

ODBC
Open Database Connectivity. A specification for an API that defines a standard set of routines with which an application can access data in a data source.

ODBC Administrator
An executable program that calls the installer DLL to configure data sources.

Open Group
A company that publishes standards. In particular, it publishes SQL Access Group (SAG) standards.

optimistic concurrency
A strategy to increase concurrency in which rows are not locked. Instead, before they are updated or deleted, a cursor checks to see if they have been changed since they were last read. If so, the update or delete fails. See also pessimistic concurrency.

outer join
A join in which both matching and nonmatching rows are returned. The values of all columns from the unmatched table in nonmatching rows are set to NULL.

owner
The owner of a table.

parameter
A variable in an SQL statement, marked with a parameter marker or question mark (?). Parameters are bound to application variables and their values retrieved when the statement is executed.

parameter descriptor
A descriptor that describes the run-time parameters used in an SQL statement, either before any conversion specified by the application (an application parameter descriptor, or APD) or after any conversion specified by the application (an implementation parameter descriptor, or IPD).

parameter operation array
An array containing values that an application can set to indicate that the corresponding parameter should be ignored in an SQLExecDirect or SQLExecute operation.

parameter status array
An array containing the status of a parameter after a call to SQLExecDirect or SQLExecute.

pessimistic concurrency
A strategy for implementing serializability, in which rows are locked so that other transactions cannot change them. See also optimistic concurrency and serializability.

positioned operation
Any operation that acts on the current row. For example, positioned update and delete statements, SQLGetData, and SQLSetPos.

positioned update statement
An SQL statement used to update the values in the current row.

positioned delete statement
An SQL statement used to delete the current row.

prepare
To compile an SQL statement. An access plan is created by preparing an SQL statement.

primary key
A column or columns that uniquely identifies a row in a table.

procedure
A group of one or more precompiled SQL statements that are stored as a named object in a database.

procedure column
An argument in a procedure call, the value returned by a procedure, or a column in a result set created by a procedure.

qualifier
A database that contains one or more tables.

query
An SQL statement. Sometimes used to mean a SELECT statement.
quoted identifier
An identifier that is enclosed in identifier quote characters so that it can contain special characters or match keywords (also known in SQL-92 as a delimited identifier).

R
radix
The base of a number system. Usually 2 or 10.
record
See row.
result set
The set of rows created by executing a SELECT statement.
return code
The value returned by an ODBC function.
roll back
To return the values changed by a transaction to their original state.
row
A set of related columns that describe a specific entity. Also known as a record.
row descriptor
A descriptor that describes the columns of a result set, either before any conversion specified by the application (an implementation row descriptor, or IRD) or after any conversion specified by the application (an application row descriptor, or ARD).
row operation array
An array containing values that an application can set to indicate that the corresponding row should be ignored in a SQLSetPos operation.
row status array
An array containing the status of a row after a call to SQLFetch, SQLFetchScroll, or SQLSetPos.
rowset
The set of rows returned in a single fetch by a block cursor.
rowset buffers
The buffers bound to the columns of a result set and in which the data for an entire rowset is returned.

S
SAG
See SQL Access Group (SAG).
scalar function
A function that generates a single value from a single value. For example, a function that changes the case of character data.
schema
See catalog.
scrollable cursor
A cursor that can move forward or backward through the result set.
serializability
Whether two transactions executing simultaneously produce a result that is the same as the serial (or sequential) execution of those transactions. Serializable transactions are required to maintain database integrity.
server database
A DBMS designed to be run in a client/server environment. These DBMSs provide a stand-alone database engine that provides rich support for SQL and transactions. They are accessed through DBMS-based drivers. For example, Oracle, Informix, DB/2, or Microsoft SQL Server.
set function
See aggregate function.
setup DLL
See driver setup DLL and translator setup DLL.
single-tier driver
See file-based driver.
SQL
Structured Query Language. A language used by relational databases to query, update, and manage data.
SQL Access Group (SAG)
An industry consortium of companies concerned with SQL DBMSs. The Open Group's Call-Level Interface is based on work originally done by the SQL Access Group.
SQL conformance level
The level of SQL-92 grammar supported by a driver; can be Entry, FIPS Transitional, Intermediate, or Full.
SQL data type
The data type of a column or parameter as it is stored in the data source.

SQLSTATE
A five-character value that indicates a particular error.

SQL statement
A complete phrase in SQL that begins with a keyword and completely describes an action to be taken. For example, SELECT * FROM Orders. SQL statements should not be confused with statements.

state
A well-defined condition of an item. For example, a connection has seven states, including unallocated, allocated, connected, and needing data. Certain operations can be done only when an item is in a particular state. For example, a connection can be freed only when it is in an allocated state and not, for example, when it is in a connected state.

state transition
The movement of an item from one state to another. ODBC defines rigorous state transitions for environments, connections, and statements.

statement
A container for all the information related to an SQL statement. Statements should not be confused with SQL statements.

statement handle
A handle to a data structure that contains information about a statement.

static cursor
A scrollable cursor that cannot detect updates, deletes, or inserts in the result set. Usually implemented by making a copy of the result set.

static SQL
A type of embedded SQL in which SQL statements are hard-coded and compiled when the rest of the program is compiled. See also dynamic SQL.

stored procedure
See procedure.

T

table
A collection of rows.

thunking
The conversion of 16-bit addresses to 32-bit addresses, or vice versa, when 16-bit applications are used with 32-bit ODBC drivers.

transaction
An atomic unit of work. The work in a transaction must be completed as a whole; if any part of the transaction fails, the entire transaction fails.

transaction isolation
The act of isolating one transaction from the effects of all other transactions.

transaction isolation level
A measure of how well a transaction is isolated. There are five transaction isolation levels: Read Uncommitted, Read Committed, Repeatable Read, Serializable, and Versioning.

translator DLL
A DLL used to translate data from one character set to another.

translator setup DLL
A DLL that contains translator-specific installation and configuration functions.

two-phase commit
The process of committing a distributed transaction in two phases. In the first phase, the transaction processor checks that all parts of the transaction can be committed. In the second phase, all parts of the transaction are committed. If any part of the transaction indicates in the first phase that it cannot be committed, the second phase does not occur. ODBC does not support two-phase commits.

type indicator
An integer value passed to or returned from an ODBC function to indicate the data type of an application variable, a parameter, or a column. ODBC defines type indicators for both C and SQL data types.

V

view
An alternative way of looking at the data in one or more tables. A view is usually created as a subset of the columns from one or more tables. In ODBC, views are generally equivalent to tables.

© 2016 Microsoft

ODBCCONF.EXE
ODBCCONF.exe is a command-line tool that allows you to configure ODBC drivers and data source names.

**Note**

ODBCCONF.exe will be removed in a future version of Windows Data Access Components. Avoid using this feature, and plan to modify applications that currently use this feature. You can use PowerShell commands to manage drivers and data sources. For more information about these PowerShell commands, see [Windows Data Access Components cmdlets](#).

**Syntax**

```
ODBCCONF [switches] action
```

**Arguments**

**switches**

Zero or more switch options. For the list of available switches, see the Remarks section, later in this topic.

**action**

One action to perform. For the list of available options, see the Remarks section.

**Remarks**

The following switches are available:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/A (action)</td>
<td>Specify an action.</td>
</tr>
<tr>
<td></td>
<td>/A is optional if only one action is specified.</td>
</tr>
<tr>
<td>/?</td>
<td>Display usage for ODBCCONF.EXE.</td>
</tr>
<tr>
<td>/C</td>
<td>Processing continues if an action fails.</td>
</tr>
<tr>
<td>/E</td>
<td>Erase the response file specified with /F when processing is finished.</td>
</tr>
<tr>
<td>/F</td>
<td>Use a response file, such as odbcconf /F my.rsp.</td>
</tr>
<tr>
<td></td>
<td>my.rsp might look like this:</td>
</tr>
<tr>
<td></td>
<td>\REGSVR c:\my.dll</td>
</tr>
<tr>
<td></td>
<td>/A is not used in a response file.</td>
</tr>
<tr>
<td>/H</td>
<td>Display usage (Help). This switch is the same as /?.</td>
</tr>
<tr>
<td>/L[mode] filename</td>
<td>Send program output to a file in one of three modes: normal (n), verbose (v), and debug (d). Debug mode records the DLLs that are loaded by odbcconf.exe.</td>
</tr>
<tr>
<td></td>
<td>If you specify /L without a mode, the log file will be empty.</td>
</tr>
<tr>
<td></td>
<td>For example, /Lv log.txt</td>
</tr>
<tr>
<td>/R</td>
<td>The action will be performed after a reboot.</td>
</tr>
<tr>
<td>/S</td>
<td>Silent mode. Do not display error messages.</td>
</tr>
</tbody>
</table>

The following actions are available:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIGDRIVER</td>
<td>Loads the appropriate driver setup DLL and calls the ConfigDriver function.</td>
</tr>
<tr>
<td>driver_name</td>
<td>Equivalent to the SQLConfigDriver function.</td>
</tr>
<tr>
<td>specific configuration params</td>
<td>For example:</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| /A (CONFIGDRIVER " Driver Name" "CPTimeout=60")  
/A (CONFIGDRIVER " Driver Name" "DriverODBCVer=03.80") | |

| CONFIGDSN **driver_name DSN=** | Adds or modifies a system data source.  
Equivalent to the SQLConfigDataSource function. | For example: |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>/A (CONFIGDSN &quot; SQL Server&quot; &quot;DSN=name</td>
<td>Server=srv&quot;)</td>
</tr>
</tbody>
</table>

| CONFIGSYSdsn **driver_name DSN=** | Adds or modifies a system data source.  
Equivalent to the SQLConfigDataSource function. | For example: |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>/A (CONFIGSYSdsn &quot; SQL Server&quot; &quot;DSN=name</td>
<td>Server=srv&quot;)</td>
</tr>
</tbody>
</table>

| INSTALLDRIVER | Equivalent to SQLInstallDriverEx Function.  
For information about the keyword-value pairs syntax passed to INSTALLDRIVER, see Driver Specification Subkeys. | For example: |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>/A (INSTALLDRIVER &quot;Your Driver</td>
<td>Driver=c:your.dll</td>
</tr>
</tbody>
</table>

| INSTALLTRANSLATOR **translator configuration path** | Adds information about a translator to the HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Translators registry  
Equivalent to SQLInstallTranslatorEx Function.  
For information about the keyword-value pairs syntax passed to INSTALLDRIVER, see Translator Specification Subkeys. | For example: |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>/A (INSTALLTRANSLATOR &quot;My Translator</td>
<td>Translator=c:my.dll</td>
</tr>
</tbody>
</table>

| REGSVR **dll** | Registers a DLL.  
Equivalent to regsvr32.exe. | For example: |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>/A (REGSVR c:my.dll)</td>
<td></td>
</tr>
</tbody>
</table>

| SETFILEDSNDIR | When HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\ODBC File DSN\DefaultDSNDir does not exist, the SETFILEDSNDIR action will c  
value at HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\CommonFilesDir, appended with \ODBC\Data Sources.  
The value at HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\ODBC File DSN\DefaultDSNDir specifies the default location used by the Administrator when creating a file-based data source. | For example: |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>/A (SETFILEDSNDIR)</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Microsoft Open Database Connectivity (ODBC)

© 2016 Microsoft
Microsoft Developer Services Agreement

Updated October, 2013

This agreement is between you or the entity you represent and the Microsoft entity listed in Exhibit A, and consists of the terms below. Exhibit A, Exhibit B, the SLAs, Offer Details for any Service as published on the date of a Service purchase or renewal, terms incorporated by reference, terms applicable to other Microsoft web sites and services that you use and are necessary to use the Services (for example, your Microsoft Account) and, the Privacy Statement (together, the “Agreement”). If you are entering into this Agreement on behalf of an entity, such as your employer, you represent that you have the legal authority to bind that entity. If you specify a company name in connection with signing up for or ordering a Service, you will be deemed to have placed that order and to have entered into this Agreement on behalf of that organization or company. Key terms are defined in Section 11. In addition, if you are a Windows Azure customer, this Agreement supplements your existing Windows Azure agreement and governs to the extent of any conflict with the Windows Azure terms (except that the Windows Azure billing terms will continue to apply).

1. Services.

a. Right to use. We grant you the right to access and use the Services in accordance with this Agreement.

b. User Plan. Each user of the Visual Studio Online portion of the Developer Services must be allocated an individual User Plan, whether they access the service directly or indirectly.

c. Manner of use. You may not:

i. reverse engineer, decompile, disassemble or work around technical limitations in the Services, except to the extent that applicable law permits it despite these limitations;

ii. disable, tamper with or otherwise attempt to circumvent any mechanism that limits your use of the Services;

iii. rent, lease, lend, resell, transfer, or sublicense any Services or portion thereof to or for third parties, except as explicitly permitted herein or in license terms that accompany any Services component;

iv. use the Services for any purpose that is unlawful or prohibited by this Agreement; or

v. use the Services in any manner that could damage, disable, overburden, or impair any Microsoft service, or the network(s) connected to any Microsoft service, or interfere with any other party’s use and enjoyment of any Services.

d. Updates. We may make changes to the Services from time to time, including: the availability of features; how long, how much or how often any given feature may be used; and feature dependencies upon other services or software. We will provide you with prior notice before removing any material feature or functionality of the Developer Services (excluding Previews), unless security, legal, or system performance considerations require an expedited removal.

e. Preview features. We may make features available on a Preview basis. Previews are provided “AS-IS” and are excluded from the SLAs and warranties in Section 7 below. Previews may be subject to reduced or different security, compliance, privacy, availability, reliability, and support commitments, as further explained in the Privacy Statement, and any additional notices provided with the Preview. We may change or discontinue Previews at any time without notice. We also may choose not to release a Preview into “General Availability”, and if we do make Previews “Generally Available” we may charge for any such features.

2. Software.

a. Using Microsoft Software outside the Service. Microsoft may provide you with Microsoft Software through or as a part of the Developer Services. Termination of use of or access to the Developer Services or the termination of this Agreement terminates your right to possess or use any such Microsoft Software; and the suspension or termination of a User Plan terminates that user’s right to possess or use any such Microsoft Software that was acquired through, is attached to, or otherwise requires that User Plan. You must delete all copies of such Microsoft Software licensed under this Agreement and destroy any associated media upon the termination of the associated possession or usage rights. Microsoft may provide you with Microsoft Software for use outside the Developer Services and with (1) the Developer Services or (2) programs you develop using the Developer Services. If the Microsoft Software is provided with its own license terms, those terms control as modified by the foregoing. If the Microsoft Software does not have its own license terms, then you may install and use any number of copies of the Microsoft Software to design, develop, and test your programs on devices. This subsection does not apply to Microsoft Software addressed in subsection (b) below.

b. Software on Documentation Portals. Software accessible on the Documentation Portals is made available by the designated publisher under the associated license terms. If Software is accessible on the Documentation Portals without license terms, then subject subsection (c) below you may use it to design, develop, and test your programs. If any such Software without license terms is marked as “sample” or “example,” then you may use it under the terms of the Microsoft Limited Public License.

c. Scope of rights. All Microsoft Software are the copyrighted works of Microsoft or its suppliers. All Microsoft Software are licensed not sold and may not be transferred unless specified otherwise in any license terms provided with the Microsoft Software. Rights to access Microsoft Software on any device do not give you any right to implement Microsoft patents or other Microsoft intellectual property in software or devices that access that device.

d. Third party software. You are solely responsible for any third party software that you install, connect, or use with any Service. We will not run or make any copies of such third party software outside of our relationship with you. You may only install or use any third party software with any Service in a way that does not subject our intellectual property or technology to any terms governing such software. We are not a party to and are not bound by any terms governing your use of any third party software. We do not grant any licenses or rights, express or implied, to such third...
Microsoft Developer Services Agreement

5. Customer accounts, customer conduct, identity services, and feedback.

a. Account creation. If any of the Services requires you to open an account, you must complete the registration process by providing us with current, complete and accurate information. You may not select an account user name or identifier that impersonates someone else, is or may be illegal, or may be protected by trademark or other proprietary rights, is vulgar or offensive or may cause confusion. We reserve the right to reject and/or reassign these user names and Service identifiers in our sole discretion.

b. Responsibility for your accounts. You are responsible for: any and all activities that occur under your account; maintaining the confidentiality of any non-public authentication credentials associated with your use of the Services; and promptly notifying our customer support team about any possible misuse of your accounts or authentication credentials, or any security incident related to the Services.
c. Your conduct and the availability of third party content and links to third party content. For any public, community interaction you undertake on the Services you must follow the Rules of Conduct. We have no obligation to monitor the content and communications of third parties on the Services; however, we reserve the right to review and remove any such materials posted to the Documentation Portals in our sole discretion. Third parties that participate on the Services are not authorized Microsoft spokespersons, and their views do not necessarily reflect those of Microsoft.

d. Identity usage across Services. We may provide Services that supplement Microsoft Software and rely upon your user account or other identity mechanism. We may use this information to identify you and authorize access to Microsoft Content, Microsoft Software, and other resources across the Services.

e. Submissions and feedback. We do not claim ownership of any Submission unless otherwise agreed to by the parties. However, by providing a Submission, you are irrevocably granting Microsoft and its Affiliates the right to make, use, modify, distribute and otherwise commercialize the Submission in any way and for any purpose (including by granting the general public the right to use your Submissions in accordance with this Agreement, which may change over time). For Submissions provided to the Documentation Portals you further grant the right to publish specific identifying information detailed in the Privacy Statement in connection with your Submission. These rights are granted under all applicable intellectual property rights you own or control. No compensation will be paid with respect to the use of your Submissions. Microsoft is under no obligation to post or use any Submission, and Microsoft may remove any Submission at any time. By providing a Submission you warrant that you own or otherwise control all of the rights to your Submission and that your Submission is not subject to any rights of a third party (including any personality or publicity rights of any person).

f. Services accessible only to invited customers. Elements of the Services may be accessible to you on an invitation only basis, for example as part of a program for using pre-release Services and providing feedback to us (e.g., through the Connect portal). Those Services are confidential information of Microsoft. You may not disclose this confidential information to any third party for a period of five years. This restriction does not apply to any information that is becoming publicly available without a breach of this restriction, was lawfully known to the receiver of the information without an obligation to keep it confidential, is received from another source who can disclose it lawfully and without an obligation to keep it confidential, or is independently developed. You may disclose this confidential information if required to comply with a court order or other government demand that has the force of law. Before doing so, you must seek the highest level of protection available and, when possible, give us enough prior notice to provide a reasonable chance to seek a protective order.

6. Term, termination, and suspension.

a. Agreement Term and termination. You may terminate this Agreement at any time. If you have purchased access to Developer Services through Windows Azure then you must pay any amounts due and owing.

b. Regulatory. In any country where any current or future government regulation or requirement that applies to us, but not generally to businesses operating there, presents a hardship to us operating the Services without change, and/or causes us to believe this Agreement or the Services may be in conflict with any such regulation or requirement, we may change the Services or terminate the Agreement. Your sole remedy for such changes to the Services under this Section is to terminate this Agreement.

c. Suspension. We may suspend your use of the Services if: (1) reasonably needed to prevent unauthorized access to Customer Data; (2) you fail to respond to a claim of alleged infringement under Sections 4.k or 8 within a reasonable time; or (3) you violate this Agreement. We will attempt to suspend access to the minimum necessary part of the Services while the condition or need exists. We will give notice before we suspend, except where we reasonably believe we need to suspend immediately. If you do not comply with the reasons for the suspension within 60 days after we suspend, we may terminate this Agreement and delete your Customer Data without any retention period. We may also terminate your account if your use of the Developer Services is suspended more than twice in any 12-month period.

d. Termination for non-usage. We may suspend or terminate a Service account after a prolonged period of inactivity. For Developer Services, if you have a free account we may terminate this Agreement and/or delete any Customer Data automatically generated during the Developer Services sign up process if you fail to upload or create any Customer Data within 90 days of your initial provisioning of the Developer Service. We will provide you with notice prior to any account suspension or termination, or Customer Data deletion.

e. Termination of Access to Documentation Portals. We reserve the right to terminate your access to the Documentation Portals at any time, without notice, for any reason whatsoever.

7. Warranties.

a. Microsoft Services warranty. If you are a Windows Azure customer who has purchased access to the Developer Services, then we warrant that the Developer Services will satisfy the SLA during the Term for the paid for portion of the Developer Services. Your only remedies for breach of this limited warranty are those in the SLA. This warranty is subject to the following limitations:

   i. any implied warranties, guarantees or conditions not able to be disclaimed as a matter of law will last one year from the start of the limited warranty;
   ii. this limited warranty does not cover problems caused by accident, abuse or use of the Developer Services in a manner inconsistent with this Agreement or our published documentation or guidance, or resulting from events beyond our reasonable control;
   iii. this limited warranty does not apply to problems caused by the failure to meet minimum system requirements; and
   iv. this limited warranty does not apply to Previews or free offerings.

OTHER THAN THIS WARRANTY, OR EXCEPT AS WARRANTED IN A SEPARATE AGREEMENT, MICROSOFT AND ITS RESPECTIVE SUPPLIERS PROVIDE THE SERVICES (INCLUDING THE CONTENT AND APIS) "AS IS," "WITH ALL FAULTS" AND "AS AVAILABLE." YOU BEAR THE RISK OF USING IT. WE PROVIDE NO WARRANTIES, GUARANTEES OR CONDITIONS, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, INCLUDING WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. YOU MAY HAVE ADDITIONAL RIGHTS UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. THESE DISCLAIMERS WILL APPLY TO THE FULLEST EXTENT PERMITTED UNDER APPLICABLE LAW, INCLUDING APPLICATION TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.

b. Third party content and materials. MICROSOFT DOES NOT CONTROL, REVIEW, REVISE, ENDORSE, OR ACCEPT RESPONSIBILITY FOR ANY THIRD PARTY CONTENT, INFORMATION, MESSAGES, MATERIALS, PROJECTS ACCESSIBLE FROM OR LINKED THROUGH THE SERVICES, AND,
EXCEPT AS WARRANTED IN A SEPARATE AGREEMENT, MICROSOFT MAKES NO REPRESENTATIONS OR WARRANTIES WHATSOEVER ABOUT AND SHALL NOT BE RESPONSIBLE FOR ANY OF THE FOREGOING. ANY DEALINGS YOU MAY HAVE WITH SUCH THIRD PARTIES ARE AT YOUR OWN RISK.


a. **Defense.** We will defend you against any claims made by an unaffiliated third party that the Developer Services or Developer Services Software infringe its patent, copyright or trademark or makes unlawful use of its trade secret. You will defend us against any claims made by an unaffiliated third party that any (1) Non-Microsoft Product that is not made available through the Developer Services or Developer Services Software or (2) Customer Data you provide directly or indirectly in using the Services infringe the third party's patent, copyright, or trademark or makes unlawful use of its trade secret.

b. **Limitations.** Our obligations in Section 8.a will not apply to a claim or award based on: (1) Customer Data, Non-Microsoft Product, modifications you make to the Services, or materials you provide or make available as part of using the Services; (2) your combination of the Services with, or damages based upon the value of, a Non-Microsoft Product, data or business process; (3) your use of a Microsoft trademark without our express written consent, or your use of the Services after we notify you to stop due to a third-party claim; or (4) your redistribution of the Services to, or use for the benefit of, any unaffiliated third party.

c. **Remedies.** If we reasonably believe that a claim under Section 8.a may bar your use of the Developer Services or Developer Services Software, we will seek to: (1) obtain the right for you to keep using it; or (2) modify or replace it with a functional equivalent. If these options are not commercially reasonable, we may terminate your rights to use the Developer Services or Developer Services Software.

d. **Obligations.** Each party must notify the other promptly of a claim under this Section 8. The party seeking protection must (1) give the other sole control over the defense and settlement of the claim; and (2) give reasonable help in defending the claim. The party providing the protection will (1) reimburse the other for reasonable out-of-pocket expenses that it incurs in giving that help and (2) pay the amount of any resulting adverse final judgment (or settlement that the other consents to). The parties’ respective rights to defense and payment of judgments or settlements under this Section are in lieu of any common law or statutory indemnification rights or analogous rights, and each party waives such common law rights.


a. **Limitation.** The aggregate liability of each party under this Agreement is limited to direct damages up to the amount paid under this Agreement for the Developer Services giving rise to that liability during the 12 months before the liability arose, or for Services provided free of charge, Five Hundred United States dollars ($500.00 USD).

b. **Exclusion.** Neither Party, nor its suppliers will be liable for loss of revenue, lost profits, or indirect, special, incidental, consequential, punitive, or exemplary damages, even if the party knew they were possible.

c. **Exceptions to Limitations.** The limits of liability in this Section apply to the fullest extent permitted by applicable law, but do not apply to: (1) the parties’ obligations under Section 8 or Exhibit A; or (2) breach of any confidentiality obligation or violation of the other’s intellectual property rights.

10. Miscellaneous.

a. **No additional rights granted.** We reserve all rights not expressly granted under this agreement, and no other rights are granted under this agreement by implication or estoppel or otherwise.

b. **Notices.** You must send notices by mail to the address listed for the Microsoft contracting entity listed in Exhibit A applicable to your primary place of business, with a copy to:

Microsoft Legal and Corporate Affairs (Developer Division)

One Microsoft Way

Redmond, WA 98052 USA

You agree to receive electronic notices from us related to the Services, which will be sent by email to your specified end user or administrator contact information or presented to you in the Service experience. Notices are effective on the date on the return receipt for mail, the date sent for email, and the date presented if within the Service experience.

c. **Assignment.** You may not assign this agreement either in whole or in part.

d. **Severability.** If any part of this agreement is held unenforceable, the rest remains in full force and effect.

e. **Waiver.** Failure to enforce any provision of this agreement will not constitute a waiver.

f. **No agency.** We are independent contractors. This agreement does not create an agency, partnership or joint venture.

g. **No third–party beneficiaries.** There are no third–party beneficiaries to this agreement.

h. **Applicable law and venue.** The choice of law and venue applicable to the geography of your primary place of business is listed in Exhibit A.

i. **Entire agreement.** This agreement is the entire agreement concerning its subject matter and supersedes any prior or concurrent communications. Additional terms applicable to this agreement based on the geography of your primary place of business are listed in Exhibit A.

j. **Survival.** The following provisions will survive this agreement’s termination: 1.b, 2.a–b, 4, 5.a–d, 5.f–g, 6, 7, 8, 9, 10, 11, Exhibit A and all other definitions.

k. **U.S. export jurisdiction.** The Services are subject to U.S. export jurisdiction. You must comply with all applicable laws, including the U.S. Export Administration Regulations, the International Traffic in Arms Regulations, and end–user, end–use and destination restrictions issued by U.S. and other governments. For additional information, see http://www.microsoft.com/exporting/.

l. **International availability.** Availability of the Services, including specific features and language versions, varies by country.

m. **Acquired rights.** You will defend us against any claim that arises from (1) any aspect of the current or former employment relationship between Microsoft Legal and Corporate Affairs (Developer Division) and the Developer Services giving rise to that liability during the 12 months before the liability arose, or for Services provided free of charge, Five Hundred United States dollars ($500.00 USD).
you and any of your current or former personnel or contractors or under any collective agreements, including, without limitation, claims for wrongful termination, breach of express or implied employment contracts, or payment of benefits or wages, unfair dismissal costs, or redundancy costs, or (2) any obligations or liabilities whatsoever arising under the Acquired Rights Directive (Council Directive 2001/23/EC, formerly Council Directive 77/187/EC as amended by Council Directive 98/50/EC) or any national laws or regulations implementing the same, or similar laws or regulations, (including the Transfer of Undertakings (Protection of Employment) Regulations 2006 in the United Kingdom) including a claim from your current or former personnel or contractors (including a claim in connection with the termination of their employment by us following any transfer of their employment to us pursuant to such laws or regulations). You must pay the amount of any resulting adverse final judgment (or settlement to which you consent). This section provides our exclusive remedy for these claims. We will notify you promptly in writing of a claim subject to this section. We must (1) give you sole control over the defense or settlement of such claim; and (2) provide reasonable assistance in defending the claim. You will reimburse us for reasonable out of pocket expenses that we incur in providing assistance.

n. Force maejore. Neither party will be liable for any failure in performance due to causes beyond its reasonable control (such as fire, explosion, power black out, earthquake, flood, severe storms, strike, embargo, labor disputes, acts of civil or military authority, war, terrorism including cyber terrorism), acts of God, acts or omissions of Internet traffic carriers, actions or omissions of regulatory or governmental bodies (including the passage of laws or regulations or other acts of government that impact the delivery of Services).

o. Modifications. We may modify this agreement at any time with or without individual notice to you by posting a revised version on the legal information section of the Developer Services and Documentation Portals (or an alternate site we identify), or by notifying you in accordance with Section 10.b. Any modifications will be effective upon your continued use of a Service.

p. Notices and procedure for making claims of copyright infringement. Pursuant to Title 17, United States Code, Section 512(c)(2), notifications of claimed copyright infringement should be sent to our designated agent. ALL INQUIRIES NOT RELEVANT TO THE FOLLOWING PROCEDURE WILL NOT RECEIVE A RESPONSE. See Notice and Procedure for Making Claims of Copyright Infringement (http://www.microsoft.com/info/cpyrtInfrg.htm).

11. Definitions.

Any reference in this agreement to “day” will be a calendar day.

“Affiliate” means any legal entity that a party owns or that owns a party, with a 50% or greater interest.

“Content” means documents, photographs, videos, and other graphical, textual, or audio-visual content that may be subject to copyright protection.

“Customer Data” means any Content or other data, including all text, sound, software, or image files that are provided to us by, or on behalf of you through your use of the Developer Services for use by you or your authorized users. Customer Data does not include Submissions or any other Content or data that you submit to the Documentation Portals or otherwise provide via the Developer Services for public access.

“Developer Services” means Visual Studio Online, the Developer Services Portal, the Visual Studio profile services, and other services we identify as governed by this Agreement.


“Developer Services Software” means Microsoft software we provide to you as part of the Developer Services for use with the Developer Services.

“Documentation Portals” means the Microsoft developer network content and marketing site available at http://msdn.microsoft.com and information technology specialist content and marketing site available at http://technet.microsoft.com, or at alternate sites we identify.

“Microsoft Content” means Content on the Services provided by Microsoft and its suppliers.

“Microsoft Limited Public License” means the Microsoft Limited Public License software license, a copy of which is provided in Exhibit B.

“Microsoft Limited Public License” means Microsoft software and computer code, including sample code and Developer Services Software.

“Non-Microsoft Product” is any software, data, service, website or other product licensed, sold or otherwise provided to you by an entity other than us, whether you obtained it via our Services or elsewhere.

“Offer Details” means the pricing and related terms applicable to paid for Developer Services.

“Preview” means preview, beta, or other pre-release versions of the Developer Services or Developer Services Software offered by Microsoft.

“Privacy Statement” means the Services privacy statement (http://go.microsoft.com/fwlink/?LinkID=246330).

“Rules of Conduct” means the Services rules of conduct (http://go.microsoft.com/fwlink/?LinkId=303819).

“Services” means the Developer Services, Documentation Portals, the http://connect.microsoft.com site, and Microsoft Software we make available to you under this Agreement.

“SLA” means the commitments we make regarding delivery or performance of the Developer Services (http://go.microsoft.com/fwlink/?LinkId=309360).

“Submissions” means Content, code, comments, feedback, suggestions, information or materials that you provide via the Documentation Portals or any Services for public access (rather than for your personal use or use by your authorized users). Submissions do not include Customer Data.

“User Plan” means a per-user based subscription, trial, or other Microsoft granted benefit that permits access to and account services for the Developer Services.

“we” and “us” means the Microsoft entity listed in Exhibit A applicable to your location and its Affiliates, as appropriate.

“you” and “your” means the person or entity accepting this Agreement to use the Services.
### Exhibit A

**Customer Location Agreement Addendum**

The Microsoft entity entering into this agreement, the applicable Microsoft entity contact information, the controlling law and venue, and additional terms governing this agreement with you are indicated in the table below for the country or region of your primary place of business.

<table>
<thead>
<tr>
<th>Microsoft Entity and Contact Information</th>
<th>Applicable Law and Venue</th>
<th>Additional Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Ireland Operations Limited</td>
<td>This agreement is governed by the laws of Ireland, without regard to its conflict of laws principles except that (1) if you are a U.S. Government entity, this agreement is governed by the laws of the United States, and (2) if you are a state or local government entity in the United States, this agreement is governed by the laws of that state. If we bring an action to enforce this agreement, we will bring it in the jurisdiction where you have your headquarters. If you bring an action to enforce this agreement, you will bring it in Ireland. This choice of jurisdiction does not prevent either party from seeking injunctive relief in any appropriate jurisdiction with respect to violation of intellectual property rights.</td>
<td></td>
</tr>
<tr>
<td>Microsoft Regional Sales Corporation</td>
<td>This agreement is governed by State of Washington law, without regard to its conflict of laws principles. Subject to sections (i) and (ii) below, if we bring an action to enforce this agreement, we will bring it in the jurisdiction where you have your headquarters. If you bring an action to enforce this agreement, you will bring it in the State of Washington, U.S.A. This choice of jurisdiction does not prevent either party from seeking injunctive relief with respect to a violation of intellectual property rights.</td>
<td>The parties agree that this Agreement be written and executed in English and that, in the event this Agreement is translated into Bahasa Indonesia to comply with the implementing regulations of Indonesian Law No. 24/2009, the English language version of this Agreement controls.</td>
</tr>
</tbody>
</table>

If your primary place of business is in Africa, Europe, or the Middle East then these terms apply to our agreement.

---

COPYRIGHT NOTICE

© 2013 Microsoft Corporation. All rights reserved.
any form of appeal or other similar recourse to a
court of law. For the purpose of this agreement
only, the People’s Republic of China does not
include Hong Kong SAR, Macau SAR and Taiwan.

If your primary place of business is in North America, South America, or all remaining regions and countries not included in the above and where the
Services are lawfully available then these terms apply to our agreement.

<table>
<thead>
<tr>
<th>Microsoft Entity and Contact Information</th>
<th>Applicable Law and Venue</th>
<th>Additional Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Corporation</td>
<td>This agreement is governed by State of Washington law, without regard to its conflict of laws principles. Any action to enforce this agreement must be brought in the State of Washington. This choice of jurisdiction does not prevent either party from seeking injunctive relief in any appropriate jurisdiction with respect to violation of intellectual property rights.</td>
<td></td>
</tr>
<tr>
<td>One Microsoft Way</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redmond, WA 98052 (États-Unis)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exhibit B

Microsoft Limited Public License

This license governs use of code marked as “sample” or “example” available on this web site without a license agreement, as provided under the section above titled "NOTICE SPECIFIC TO SOFTWARE AVAILABLE ON THIS WEB SITE." If you use such code (the “software”), you accept this license. If you do not accept the license, do not use the software.

1. Definitions
The terms “reproduce,” “reproduction,” “derivative works,” and “distribution” have the same meaning here as under U.S. copyright law.

A “contribution” is the original software, or any additions or changes to the software.

A “contributor” is any person that distributes its contribution under this license.

“Licensed patents” are a contributor’s patent claims that read directly on its contribution.

2. Grant of Rights
(A) Copyright Grant – Subject to the terms of this license, including the license conditions and limitations in section 3, each contributor grants you a non-exclusive, worldwide, royalty-free copyright license to reproduce its contribution, prepare derivative works of its contribution, and distribute its contribution or any derivative works that you create.

(B) Patent Grant – Subject to the terms of this license, including the license conditions and limitations in section 3, each contributor grants you a non-exclusive, worldwide, royalty-free license under its licensed patents to make, have made, use, sell, offer for sale, import, and/or otherwise dispose of its contribution in the software or derivative works of the contribution in the software.

3. Conditions and Limitations
(A) No Trademark License- This license does not grant you rights to use any contributors’ name, logo, or trademarks.

(B) If you bring a patent claim against any contributor over patents that you claim are infringed by the software, your patent license from such contributor to the software ends automatically.

(C) If you distribute any portion of the software, you must retain all copyright, patent, trademark, and attribution notices that are present in the software.

(D) If you distribute any portion of the software in source code form, you may do so only under this license by including a complete copy of this license with your distribution. If you distribute any portion of the software in compiled or object code form, you may only do so under a license that complies with this license.

(E) The software is licensed “as-is.” You bear the risk of using it. The contributors give no express warranties, guarantees or conditions. You may have additional consumer rights under your local laws which this license cannot change. To the extent permitted under your local laws, the contributors exclude the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

(F) Platform Limitation – The licenses granted in sections 2(A) and 2(B) extend only to the software or derivative works that you create that run on a Microsoft Windows operating system product.