Application Programmer's Guide

Contents

Introduction

The ODBC (Open Database Connectivity) interface is a C programming language interface for database connectivity. The *ODBC Application Programmer's Guide* is designed to address the following three questions:

- What is the ODBC interface?
- What features does the interface offer?
- How do applications use the interface?

The guide is organized into the following chapters:

- Chapter 1, “ODBC Theory of Operation,” provides conceptual information about the ODBC interface.
- Chapter 2, “A Short History of SQL,” contains a brief history of SQL.
- Chapters 3 through 8 describe how and when to call ODBC functions.
- Chapter 9, “Constructing an ODBC Application,” lists developer's kit contents, describes how to build an application, and includes sample code.

Appendix A contains SQL syntax information.

For information about the syntax and semantics of each ODBC function, refer to the *ODBC API Reference*. For information about driver development, refer to the *ODBC Driver Developer's Guide*.

Audience

The ODBC software development kit is available for use with the C language in a Windows environment. Use of the ODBC interface spans four areas of knowledge: SQL statements, ODBC function calls, C programming, and Windows programming. This manual assumes the following expertise:

- Working knowledge of the C programming language.
- General database knowledge and a familiarity with SQL.

Document Conventions

This manual uses the following typographic conventions.

Format	Used for
WIN.INI	Names of applications, programs, and other files.
RETCODE SQLFetch (hDBC)	Sample command lines and program code.
argument	Information that you or the application must provide, or word emphasis.
SQLTransact	Syntax that must be typed exactly as shown, including function names.
[]	Optional items or, if in bold text, brackets that must be included in the text string.
|	Separates two mutually exclusive choices in a syntax line.
{}	Delimits mutually exclusive choices in a syntax line.
...	Arguments that can be repeated several times.

Where to Find Additional Information

XE "SQL:references for additional information"§For more information about driver development, refer to the *ODBC Driver Developer's Guide*, included in the ODBC Developer's Kit.

For more information about SQL, the following standards are available:

- Database Language - SQL with Integrity Enhancement, ANSI, 1989 ANSI

X3.135-1989.

- X/Open and SQL Access Group SQL CAE draft specification (1991).
- Database Language SQL: ANSI X3H2 and ISO/IEC JTC1,SC21,WG3 (draft international standard).

In addition to standards and vendor-specific SQL guides, there are many books that describe SQL, including:

- Date, C. J.: *A Guide to the SQL Standard* (Addison-Wesley, 1989).
- Emerson, Sandra L., Darnovsky, Marcy, and Bowman, Judith S.: *The Practical SQL Handbook* (Addison-Wesley, 1989).
- Groff, James R. and Weinberg, Paul N.: *Using SQL* (Osborne McGraw-Hill, 1990).
- Gruber, Martin: *Understanding SQL* (Sybex, 1990).
- Hursch, Jack L. and Carolyn J.: *SQL, The Structured Query Language* (TAB Books, 1988).
- Pascal, Fabian: *SQL and Relational Basics* (M & T Books, 1990).
- Trimble, J. Harvey, Jr. and Chappell, David: *A Visual Introduction to SQL* (Wiley, 1989).
- Van der Lans, Rick F.: *Introduction to SQL* (Addison-Wesley, 1988).
- Vang, Soren: *SQL and Relational Databases* (Microtrend Books, 1990).
- Viescas, John: *Quick Reference Guide to SQL* (Microsoft Corp., 1989).

1 ODBC Theory of Operation

XE "ODBC"§The Open Database Connectivity (ODBC) interface allows applications to access data from database management systems (DBMS).

XE "Interoperability"§The interface permits maximum *interoperability*—a single application can access diverse database management systems. You can develop, compile, and ship an application without targetting a specific DBMS. Users can then add modules called database *drivers* that link your application to their choice of database management systems.

The ODBC interface defines the following:

- A library of ODBC function calls that allow an application to connect to a DBMS, execute SQL statements, and retrieve results.
- SQL syntax. The syntax is based on the X/Open and SQL Access Group SQL CAE draft specification (1991).. To send an SQL statement, you include the statement as an argument in an ODBC function call. You need not customize the statement for a specific DBMS.

  Appendix A, "SQL Grammar," contains a SQL syntax based on the X/Open and SQL Access Group SQL CAE draft specification (1991). To ensure maximum interoperability, ODBC applications are encouraged to use only the SQL syntax defined in Appendix A.
- A standardized set of error codes.
- A standard way to connect and log on to a DBMS.
- A standardized representation for data types.

The interface is flexible:

- You can construct SQL statements at compile time or at run time.
- You can use the same object code to access different DBMS products.
- You can connect to multiple instances of DBMS products.
- Your application can ignore underlying data communications protocols between your application and a DBMS product.
- You can send and retrieve data values in a format convenient to the application.

XE "ODBC:function call types"§The ODBC interface provides two types of function calls:

- Core functions based on the X/Open and SQL Access Group Call Level Interface specification.

- Extended functions support additional functionality, including scrollable cursors and asynchronous processing.

An application can check for availability of functions at run-time.

The following code fragment shows one way to use the ODBC interface to submit an SQL CREATE TABLE statement:

```
UCHAR sqltext[] ="CREATE TABLE NAMEID (ID integer, NAME varchar(50))";
if(SQLExecDirect(hstmt, sqltext, SQL_NTS) !=SUCCESS)
return(print_err(hDBC,hstmt));
```

Note that an application could obtain the value of sqltext from a dialog box at run time.

The following sections describe the ODBC architecture in more detail.

ODBC Components

The following paragraphs describe components within the ODBC architecture. This information is provided for reference purposes; the boundaries between the three underlying components are transparent to an application.

XE "ODBC:components"§The ODBC architecture has four components:

- Application Performs processing, possibly on behalf of a user, and calls ODBC functions to submit SQL statements and retrieve results.
- Driver Manager Loads drivers on behalf of an application.
- 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.

The Driver Manager and driver appear to an application as one unit that processes ODBC function calls. The following diagram shows the relationship between the four components. The following paragraphs describe each component in more detail.

Application

XE "Application"§An application that uses the ODBC interface performs the following tasks:

- Requests a connection, or session, with a data source.
- Sends SQL requests to the data source.
- Defines storage areas and data formats for the results of SQL requests.
- Requests results.
- Processes errors.
- Reports results back to a user, if necessary.
- Requests commit or rollback operations for transaction control.
- Terminates the connection to the data source.

An application can provide a variety of features external to the ODBC interface, including spreadsheet capabilities, online transaction processing, and report generation; the application may or may not interact with users.

Driver Manager

XE "Driver Manager"§The Driver Manager, provided by Microsoft, is a dynamically-linked library (DLL) with an import library. The primary purpose of the Driver Manager is to load drivers. (The following section describes drivers.) In addition to loading drivers, the Driver Manager performs the following:

- Uses the ODBC.INI file to map a data source name to a specific driver dynamically-linked library (DLL).
- Processes several initialization-oriented ODBC calls.
- Provides entry points to ODBC functions for each driver.
- Provides parameter validation and sequence validation for ODBC calls.

Driver

XE "Driver"§A driver is a DLL that implements ODBC function calls and interacts with a data source

A driver is loaded by the Driver Manager when the application calls the **SQLConnect** or **SQLDriverConnect** function.

A driver performs the following tasks in response to ODBC function calls from an application:

- Establishes a connection to a data source.

- Submits requests to the data source.
- Translates data to or from other formats, if requested by the application.
- Returns results to the application.
- Formats errors into standard error codes and returns them to the application.
- Declares and manipulates cursors if necessary. (This operation is invisible to the application unless you request access to a cursor name.)
- Initiates transactions if the data source requires explicit transaction initiation. (This operation is invisible to the application.)

Data Source

XE "Database environment"§This manual refers to the general features and functionality provided by an SQL database management system as a DBMS. The manual refers to a specific instance of a combination of a DBMS product, remote operating system, and network as a data source.

An application establishes a connection with a particular vendor's DBMS product on a particular operating system, accessible by a particular network. For example, the application might establish connections to:

- An Oracle DBMS running on an OS2 operating system, accessed by Novell netware.
- A local xBase file, in which case the network and remote operating system are not part of the communication path.
- A Tandem NonStop SQL DBMS running on the Guardian 90 operating system, accessed via a gateway.

Types of Drivers

XE "Driver:types"§ODBC defines two types of drivers:

- Single-tier The driver processes both ODBC calls and SQL statements. (In this case, the driver performs part of the data source functionality.)
- Multiple-tier The driver processes ODBC calls and passes SQL statements to the data source.

One system can contain both types of configurations.

The following paragraphs describe single-tier and multiple-tier configurations in more detail.

Single-Tier Configuration

XE "Configuration:single-tier"§In a single-tier implementation, the database is a file and is processed directly by the driver. The driver processes SQL statements and retrieves information from the database. One example of a single-tier implementation is a driver that manipulates an xBase file.

The set of SQL statements that you can submit may be limited by a single-tier driver. The minimum set of SQL statements that must be supported by a single-tier driver is defined in Appendix A, "SQL Grammar."

The following diagram shows two types of single-tier configurations.

Multiple-Tier Configuration

XE "Configuration:multiple-tier"§In a multiple-tier configuration, the driver sends SQL requests to a server that processes SQL requests.

The application, driver, and Driver Manager reside on one system, typically called the client. The database and the software that controls access to the database typically reside on another system, typically called the server.

One type of multiple-tier configuration is a gateway architecture where the driver passes SQL requests to a gateway process. The gateway process sends the requests to the data source.

The following diagram shows three types of multiple-tier configurations. From the perspective of an application, all three configurations are identical.

Network Example

The following diagram shows how each of the preceding configurations could appear in a single network. The diagram includes examples of the types of DBMS that could reside in a network.

Applications can also communicate across wide-area networks:

Matching an Application to a Driver

XE "ODBC:functionality sets"§One of the strengths of the ODBC interface is interoperability: you can create your ODBC application without targeting a specific data source. Users can add drivers to your application after you compile and ship the application.

From an application standpoint, it would be easiest if every driver and data source supported the same set of ODBC function calls and SQL statements. However, drivers and associated data sources provide a varying range of functionality. Therefore, the ODBC interface defines conformance levels, which determine the functions and SQL statements supported by a driver.

ODBC Function Call Support

Each ODBC driver supports a set of core ODBC functions and, optionally, one or more extended functions or data types, defined as extensions:

- Core functions and data types are based on the X/Open and SQL Access Group Call Level Interface specification. If a driver supports all core functions, it is said to conform to X/Open and SQL Access Group core functionality. If any core functions are not supported, the driver does not conform to X/Open and SQL Access Group core functionality. The specific set of core data types supported by a driver depends upon the set of data types supported by the data source.
- Extension functions and data types support additional features, including date, time, and datetime data type literals, scrollable cursors, retrieval of data dictionary information, and asynchronous execution of function calls. Extended functions may not be supported by a specific driver. The **SQLGetFunctions** function returns information about supported functions. **SQLGetFunctions** is always available to applications.

  Extended functions are divided into two conformance levels, Level 1 and Level 2, each of which is a superset of the core functions.

For a list of functions and their conformance levels, refer to the *ODBC API Reference,* Chapter 1, "ODBC Function Summary."

§

Note Each function description in this manual indicates whether the function is a core function or a level 1 or level 2 extension.

§

SQL Statement Support

Each ODBC driver supports one of two sets of SQL statements:

- The Minimum set is a set of SQL statements that can be implemented by single-tier drivers.
- The Core set is based on the X/Open and SQL Access Group SQL draft CAE specification (1991).

In addition to the core and minimum sets, ODBC defines SQL syntax for data literals, outer joins, and SQL scalar functions. For more information about SQL statement sets, refer to Appendix A, "SQL Grammar."

The grammar listed in Appendix A is not intended to restrict the set of statements that can be supported. A driver can support additional syntax that is unique to the associated data source.

How to Select a Set of Functionality

The decision to use a set of functionality depends on:

- The set of features you want to access from your application, including features that may not be available from all data sources.
- How much interoperability you would like to provide.
- How much conditional code you want to include to determine whether a function or data type is supported by the driver.
- Performance requirements. The choice of performance options increases from the minimum to the extended set, depending on data source support.

To communicate with a specific driver and data source, select the appropriate functionality set for the driver. If you want additional interoperability, the following tables may help you select a functionality set.

If you want to communicate with	Choose
Single-tier and multiple-tier drivers, with maximum interoperability and the least amount of application work	Minimum functionality. All drivers support core ODBC functions and minimum SQL statements.
Single-tier and multiple-tier drivers, with maximum interoperability and maximum functionality	Check before you issue core or extended functions. If supported, use them. If not, perform equivalent work using minimum functions.
Single-tier drivers	Minimum functionality.
Multiple-tier drivers only, with maximum interoperability and the least amount of application work	Core functionality.
Multiple-tier drivers only, with maximum interoperability, maximum functionality, and maximum performance	Extended functionality. Check functions and, if not available, perform equivalent work using core functions.

Chapters 3 through 9 describe how to develop an application that uses ODBC functions. The *ODBC API Reference* lists all ODBC functions in alphabetic order.

Connections and Transactions

Before your application can communicate with a data source, you must establish a connection. If the connection is successful, the driver returns a connection handle (a pointer to a storage area) for use in subsequent ODBC calls.

The ODBC interface allows you to request multiple connections for one or more data sources. Each connection is considered a separate transaction space.

An active connection can have one or more statement processing streams.

XE "Transactions"§XE "Connection handles:and transactions"§Each active connection maintains a transaction in progress. You can request that the results of each SQL statement be automatically committed upon completion or you can choose to explicitly request commit or rollback operations. When an application commits or rolls back a transaction, the driver resets all statement requests associated with the connection.

2 A Short History of SQL

XE "SQL, overview"§This chapter provides a brief history of SQL and describes programmatic interfaces to SQL. For more information about SQL, refer to the references listed in the introduction.

SQL Background Information

SQL, or Structured Query Language, is a widely accepted industry standard for data definition, data manipulation, data management, access protection, and transaction control. SQL originated from the concept of relational databases and uses tables, indexes, keys, rows, and columns to identify storage locations.

Many types of applications use SQL statements to access data. Examples include ad-hoc query facilities, decision support applications, report generation utilities, and online transaction processing systems.

SQL is not a complete programming language in itself. For example, there are no provisions for flow control.

One of the challenges during the evolution of SQL has been to provide a standard access to SQL database management systems from traditional programming languages like C, COBOL, and PL/1.

ANSI 1989 Standard

XE "ANSI standards"§SQL was first standardized by the American National Standards Institute (ANSI) in 1986. The first ANSI standard defined a language that was independent of any programming language.

XE "Module language"§XE "Embedded SQL (ANSI)"§XE "SQL statements:embedded"§XE "Direct invocation (ANSI)"§The first ANSI standard has since been refined; the current standard is ANSI 1989. The ANSI 1989 standard defines three programmatic interfaces to SQL:

- Module language Allows you to define procedures within compiled programs (modules). You then call these procedures from traditional programming languages. The module language uses parameters to return values to the calling program.
- Embedded SQL Allows you to embed SQL statements within your program. The specification defines embedded statements for COBOL, FORTRAN, Pascal, and PL/1.
- Direct invocation Access is implementation-defined.

Neither the module language nor the direct invocation approach has been widely implemented; most implementations use the embedded approach.

Embedded SQL

Embedded SQL allows you to place SQL statements into a program that is written in a traditional programming language (for example, COBOL or Pascal). You delimit SQL statements with specific starting and ending statements defined by the host language. The resulting program contains source code from two languages—SQL and the host language.

When you compile a program with embedded SQL statements, you use a precompiler to compile the SQL statements. The precompiler replaces the SQL statements with equivalent host language source code. After you precompile the program, you use your host language compiler to compile the resulting source code.

XE "Static SQL"§XE "SQL statements:static"§The term *static SQL* encompasses the basic features of embedded SQL. Static SQL has the following characteristics:

- To use static SQL, you define each SQL statement within the source code of your program. You specify the number of result columns and their data types before you compile your program.
- Variables called *host variables* are accessible to both your host-language code and your SQL requests. You cannot, however, use host variables for column names or table names. In addition, host variables are fully defined (including length and data type) prior to compilation.
- If you submit an SQL request that returns more than one row of data, you define a *cursor* that points to one row of result data at a time.
- Each run of the associated program performs exactly the same SQL request, with possible variety in the values of host variables. All table names and column names must remain the same from one execution of the program to the next; otherwise, you must recompile the program.
- You use standard data storage areas for status and error information.

Static SQL is efficient; you can precompile SQL statements prior to execution and run them multiple times without recompiling the statements. The application is bound to a particular DBMS when it is compiled.

Static SQL cannot defer the definition of the SQL statement until run-time. Therefore, static SQL is not the best option for client-server configurations or for ad-hoc requests.

Future ANSI Specifications

XE "SQL2, overview"§SQL2 is the most recent ANSI specification, and is in the final stages of becoming an international standard. SQL2 defines three levels of functionality: entry, intermediate, and full. SQL2 adds many new features, including:

- Additional data types, including date and time.
- Connections to database environments, to address the needs of client-server architectures.
- Support for dynamic SQL (described in the following subsection).
- Scrollable cursors for access to result sets (full level).
- Outer joins (intermediate and full levels).

Dynamic SQL

XE "Dynamic SQL"§XE "SQL statements:dynamic"§Dynamic SQL allows an application to generate and execute SQL statements at run time.

You can prepare dynamic SQL statements. When you prepare a statement, the database environment generates an access plan and a description of the result set. You can then execute the statement multiple times with the previously-generated access plan, which minimizes processing overhead.

XE "Parameters:markers"§You can include parameters in dynamic SQL statements. Parameters function in much the same way as host variables in embedded SQL. Prior to execution, you assign values to the place held by each parameter. Unlike static SQL, parameters do not require length or data type definition prior to program compilation.

Dynamic SQL is not as efficient as static SQL, but is very useful if an application requires:

- Flexibility to construct SQL statements at run time.
- Flexibility to defer an association with a database until run time.

Call Level Interface

XE "Call level interface"§XE "CLI"§A Call Level Interface (CLI) for SQL consists of a library of function calls that support SQL statements. The ODBC interface is a CLI.

The ODBC interface is designed to be used directly by application programmers, and not as the target of a preprocessor for embedded SQL.

A CLI is very straightforward to programmers who are familiar with function libraries. The function call interface does not require host variables or other embedded SQL concepts.

A CLI does not require a precompiler. To submit an SQL request, you place an SQL command into a text buffer and pass the buffer as a parameter in a function call. CLI functions provide declarative capabilities and request management. You obtain error information as you would for any function call—by return code or error function call, depending on the CLI.

XE "Binding result columns"§A CLI allows you to specify result storage before or after the results are available. This allows you to determine what the results are and take appropriate action without being limited to a specific set of data structures that were defined prior to the request. Deferral of storage specification is called late binding of variables.

The concept of a CLI is very useful in a client/server environment; the interface between the application and the data source can be designed to minimize network traffic.

A CLI is typically used for dynamic access because applications that use a CLI are often driven by user input. The CLI defined by the X/Open and SQL Access Group—and therefore the ODBC interface—are similar to the dynamic embedded version of SQL described by in X/Open and SQL Access Group draft specification "Structured Query Language (SQL)" (1991).

For a comparison between embedded SQL statements and the ODBC call level interface, refer to the *ODBC API Reference*, Appendix E, “Comparison Between Embedded SQL and ODBC .”

Interoperability

XE "Interoperability"§Interoperability for call-level interfaces can be addressed in

the following ways:

- All clients and data sources adhere to a standard interface.
- All clients adhere to a standard interface; driver programs interpret the commands for a specific data source.

The second approach allows drivers to shield clients from database functionality differences, database protocol differences, and network differences. ODBC follows the second approach. ODBC can take advantage of standard database protocols and network protocols, but does not require the use of a standard database protocol or network protocol.

3 Guidelines for Calling ODBC Functions

This chapter describes characteristics of ODBC functions and discusses how to perform the following tasks:

- Calling ODBC functions.
- Submitting SQL statements from ODBC functions.
- Handling ODBC results.

Calling ODBC Functions

The following paragraphs describe general characteristics of ODBC functions.

General Information

XE "ODBC:functions, general information"§Each ODBC function name starts with the prefix SQL. Each function accepts one or more arguments. Arguments are defined for input (to the driver) or output (from the driver).

The initialization file must contain driver location information. For additional information about initialization file contents, refer to Chapter 9, “Constructing an ODBC Application.”

C programs that call ODBC functions must use header files that define constants, type definitions, and function prototypes for all ODBC functions. To view the SQL.H and SQLEXT.H header files, refer to the *ODBC API Reference*, Appendix F, “ C Header Files.”

For a list of valid data types, refer to the *ODBC API Reference*, Appendix D, “Data Type Definitions.”

Variable Length Data in Function Arguments

XE "Variable-length data, in arguments"§XE "Arguments:variable-length data and"§All function arguments that point to variable length data (for example, column names and parameter values) have an associated length argument.

You can specify one of the following lengths XE "SQL_NULL_DATA"§XE "SQL_NTS"§for each input argument:

- A length greater than or equal to zero specifies the actual length. A length of zero describes a zero length string, which is distinct from a NULL value.

- A length equal to SQL_NULL_DATA specifies a null parameter value.
- A length equal to SQL_NTS specifies that a value is a null terminated string.

Nulls are always valid for output pointers, unless otherwise noted in the syntax description for a function.

The application is responsible for allocating memory for output buffers. Therefore, the application must indicate the length of each buffer. On output, the driver returns the actual length of data that was stored. For each output argument there are two length arguments:

- An input argument that contains the buffer length as allocated by the application, including one byte for a null termination character that the driver returns for arguments that contain character data.

  If a string argument is null for an input parameter, the driver ignores the argument unless it is required for proper operation of the function. If required, as in **SQLPrepare**, the driver returns an error. Nulls are always valid for output pointers, unless noted otherwise in the syntax description for a function.

- An output argument that contains the actual number of bytes written to the buffer by the driver (not including the null termination character), or SQL_NULL_DATA, if null.

  If the data does not fit in the output buffer, the driver stores the number of bytes available and returns the value SQL_SUCCESS_WITH_INFO. If the application calls **SQLError**, the driver returns a truncation error. The application can compare the output length with the buffer size to determine which value was truncated.

When converting a binary (hexadecimal) SQL data type to a character format, the output length will always be an even number of bytes.

There may be instances—for example, if the buffer is very large—when you would like to use a null pointer for the output length, then search for the termination character in the buffer. If you use null terminated strings, you can pass a null pointer for the output length, in which case the driver does not return the length. This is not recommended, however, because the driver cannot return a truncation indicator if you use a null pointer.

For more information about error names and other predefined constants, refer to the header file listed in the *ODBC API Reference*, Appendix F, “ C Header Files.”

Environment, Connection, and Statement Handles

XE "Handles:connection and statement"§XE "Connection handles"§XE "Statement handles"§To communicate with a data source, an application establishes a connection with the driver. The driver returns handles that reference data structures that store information pertinent to the ODBC environment, a specific connection to an instance of a data source, or a statement being sent to an instance of a data source. These handles are required by most ODBC functions.

the ODBC interface defines three types of handles:

- Environment handles identify memory storage for global information, including valid connection handles and current active connection handle. ODBC defines environment handles as variables of type HENV. An application must request an environment handle prior to connecting to a data source.
- Connection handles identify memory storage for information about a particular connection. ODBC defines connection handles as variables of type HDBC. An application must request a connection handle prior to a connection to a specific instance of a data source.
- Statement handles identify memory storage for information about an SQL statement. ODBC defines statement handles as variables of type HSTMT. An application must request a statement handle prior to submitting SQL requests. Each statement handle is associated with exactly one connection handle. Each connection handle can, however, have multiple statement handles associated with it.

For more information about requesting a connection handle, refer to Chapter 5, “Establishing Connections.” For more information about requesting a statement handle, refer to Chapter 6, “Preparing and Executing an SQL Statement.”

Submitting SQL Statements

XE "SQLPrepare:submitting SQL statements"§XE "SQLExecDirect:submitting SQL statements"§XE "SQL statements:processing"§To submit an SQL statement, you pass it as an argument in an ODBC function call. For more information about submitting SQL statements, refer to Chapter 6, “Preparing and Executing an SQL Statement."

The application is responsible for submitting correct SQL syntax.

For a description of grammar that is valid in ODBC function calls, refer to Appendix A, “SQL Grammar.” For a comparison between embedded SQL statements and ODBC function calls, refer to the *ODBC API Reference*, Appendix E, “Comparison Between Embedded SQL and ODBC.”

Data Type Support

XE "Data types"§The ODBC interface defines two sets of data types:

- ODBC data types define the SQL data type in the data source. This set is further divided into two subsets:
  - Core data types provide a standard set of data types. If you use the core set of functionality, the driver maps all data to a core data type. For example, the driver stores a date as a character string.

- Extended data types support data types in many current DBMS products.
- C data types describe how data is stored in your C program.

Each ODBC data type has a corresponding C data type. These data types are defined in the ODBC header files. For a list of ODBC data types, their meanings, and how they correspond to C data types, refer to the *ODBC API Reference*, Appendix D, “Data Type Definitions.” For information about the header files, refer to the *ODBC API Reference*, Appendix F, "C Header Files."

XE "SQLDescribeCol:retrieving data types"§To retrieve the underlying data type for a column, call **SQLDescribeCol**.

XE "SQLGetTypeInfo:retrieving supported data types"§The ODBC interface provides support for any data type from any data source if you retrieve the type code for the data. If you access extended functions, call **SQLGetTypeInfo** to retrieve a description of data types supported by the data source.

Handling Results

XE "Return codes"§XE "SQL_SUCCESS return code"§XE "SQL_SUCCESS_WITH_INFO return code"§XE "SQL_NO_DATA_FOUND return code"§XE "SQL_ERROR return code"§XE "SQL_INVALID_HANDLE return code"§When you call an ODBC function, the driver returns a predefined status code that indicates success or failure. The status codes indicate success, warning, or failure status. The application can then call **SQLError**, if necessary, to retrieve additional information. The following table lists return constants.

#define name	Description
SQL_SUCCESS	Function completed successfully; no additional information is available.
SQL_SUCCESS_WITH_INFO	Function completed successfully. Call **SQLError** to retrieve a warning or additional information.
SQL_NO_DATA_FOUND	All rows from the result set have been fetched.
SQL_ERROR	Function failed. Call **SQLError** for more information.
SQL_INVALID_HANDLE	Function failed due to an invalid connection handle or statement handle. This indicates a programming error. No further information is available from **SQLError**.
SQL_STILL_EXECUTING	A function was called asynchronously and is still executing.
SQL_NEED_DATA	While processing an SQL statement, the driver determined that the application needs to send large data values.

Handling Errors

XE "SQLError"§If an ODBC function other than **SQLError** returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, call **SQLError** to obtain additional information. Additional error or status information can come from one of two sources:

- Error or status information from an ODBC function, indicating that a programming error was detected.
- Error or status information from the data source, indicating that an error occurred during SQL statement processing.

The driver buffers errors or messages for only one ODBC call at a time; a subsequent call overwrites existing error information.

SQLError never returns error information about itself.

If you are familiar with SQLSTATE in the X/Open and SQL Access Group "Structured Query Language (SQL)" CAE draft specification (1991), note that the information provided by **SQLError** is in the same format as that provided by SQLSTATE.

For more information about error codes, refer to the *ODBC API Reference*, Appendix A, “ODBC Error Codes.”

Processing Result Sets

You can use the following mechanisms to retrieve information about results:

- The return code.
- A call to **SQLRowCount**.
- A call to **SQLNumResultCols**.
- A call to **SQLDescribeCol**.
- A call to **SQLColAttributes**.
- A call to **SQLNumParams**.

XE "Processing results"§XE "Results:processing"§If the operation does not affect or return rows, such as an SQL **GRANT** or **REVOKE** operation, check the return code to determine the outcome of the operation. If the operation affected rows, obtain the row count to determine the outcome of the operation. If your request was a **SELECT** query, check the number of result columns and data descriptions to gain information about the result set.

For more information about retrieving data, refer to Chapter 7, “Retrieving Results.”

4 Basic Application Steps

To use ODBC functions, perform the following steps:

1. Establish a connection to the appropriate driver. You specify a data source name, defined in the initialization file, and additional connection information.
2. For each SQL request:
   - Place the SQL text string into a buffer. Your request can include embedded parameter markers. Set parameter values if necessary. Associate a cursor name with the request if desired; otherwise, the driver assigns a cursor name. The driver requests cursor operations automatically.
   - Submit the SQL string for execution. The ODBC interface provides separate function calls for prepared and immediate execution.
   - Inquire about the results. For example, for a **SELECT** query you can request information about the columns in the result set. You can then allocate storage and define the data format for each result column.

     For unsuccessful requests, process error information.
   - Fetch data row by row.
3. When you finish a transaction, perform a commit or rollback command if you performed **INSERT**, **UPDATE**, or **DELETE** operations.
4. When you finish submitting statements to the data source, terminate the connection.

The following diagram shows an example of the basic command flow for connecting to a data source, processing SQL statements, and disconnecting from the data source. The words starting with SQL are ODBC function call names.

Depending on the types of requests your application makes, you may decide to use additional ODBC functions.

Additional Information

Chapters 5 through 8 describe how to use ODBC functions that provide these services.

The *ODBC API Reference* lists syntax and usage information for each ODBC function.

5 Establishing Connections

XE "Connections:establishing"§This chapter describes how to establish a connection to a target data source.

Using the Driver Manager

XE "Driver Manager:communicating with"§The Driver Manager is a DLL that provides access to ODBC drivers. You do not need to call an ODBC function to initiate communication with the Driver Manager; the Driver Manager is automatically involved in all ODBC calls.

Whenever you call an ODBC function, the Driver Manager performs one of the following actions:

- For **SQLAllocEnv**, **SQLAllocConnect**, **SQLDataSources**, **SQLFreeConnect**, or **SQLFreeEnv**, the Driver Manager processes the call.
- For **SQLConnect**, **SQLDriverConnect**, **SQLError**, or **SQLGetFunctions**, the Driver Manager performs initial processing then sends the call to the driver associated with the connection.
- For any other ODBC function, the Driver Manager passes the call to the driver associated with the connection.

Initializing the ODBC Environment

The **SQLAllocEnv** function initializes the ODBC interface for use by an application. **SQLAllocEnv** must be called prior to any other ODBC function:

1. Declare a variable of type HENV. For example, you could declare a variable called henv1: “HENV henv1.” HENV is defined in the SQL.H ODBC header file. For Windows, the HENV type is a memory handle. For more information about the header file, refer to the *ODBC API Reference,* Appendix F, “C Header Files.”
2. Pass the address of this variable as an argument in a call to **SQLAllocEnv**. The driver allocates storage for environment information and places the address of the storage into the HENV variable.

These steps needs to be performed only once by an application; **SQLAllocEnv** supports one or more connections to data sources.

Establishing a Connection to a Data Source

First, request a connection handle:

1. Declare a variable of type HDBC. For example, you could declare a variable called hdbc1: “HDBC hdbc1.” HDBC is defined in the ODBC header file. For Windows, the HDBC type is a memory handle.
2. Pass the address of this variable as an argument in a call to **SQLAllocConnect**. The driver allocates storage for connection information and places the address of the storage into the HDBC variable.

Next, specify a specific driver and data source. Pass the following information to the driver in a call to **SQLConnect**:

- Data source name The name of the data source being requested by the application. For Windows, this corresponds to an entry in the ODBC initialization file (ODBC.INI). For more information, refer to Chapter 9, "Constructing an ODBC Application."
- User ID The login ID or account name for access to the data source, if appropriate (optional).
- Authentication string (password) A character string associated with the user ID that allows access to the data source (optional).

The Driver Manager establishes a connection with the specified data source and returns connection status to the application.

Accessing ODBC Functions

The ODBC interface defines two types of conformance:

- SAG core conformance, which is met if a driver supports all core functions and false if the driver does not support one or more core functions.
- ODBC conformance, which has two levels:
  - Level 1. The driver supports all core functions plus an additional set of functions that provides a basic level of support and optimization for an interactive query application.
  - Level 2. The driver supports all core functions and all ODBC functions.

If the driver does not support Level 1 or Level 2 functionality, it must return "None" for ODBC conformance, but may support one or more extended functions. An application can call **SQLGetFunctions** to determine if the driver supports a particular function.

The *ODBC API Reference*, Chapter 1, "ODBC Function Summary," lists

conformance levels for all functions. In addition, all function descriptions in the reference manual indicate whether a function is a core function or a level 1 or level 2 extension.

In addition, you can call **SQLGetInfo** to determine the conformance level supported by a driver.

Extensions for Establishing Connections

Several extended functions support the connection process. The following table lists these functions in alphabetic order. The paragraphs following the table describe **SQLDriverConnect** in more detail.XE "SQLDataSources"§XE "Data source:listing"§XE "Timeout values, setting"§XE "Rowcount, setting maximum"§XE "Autocommit, setting"§XE "Transactions:setting autocommit option"§XE "SQLSetConnectOption"§XE "SQLSetStmtOption"§XE "SQLGetInfo"§XE "SQLGetTypeInfo"§

Function Name	Description
SQLDataSources	Requests a list of available data sources. The Driver Manager retrieves this information from the ODBC.INI file. You can present this information to a user or select a database and driver combination from within your application.
SQLDriverConnect	Ask the Driver Manager to present a connection dialog box to the user.
SQLGetFunctions	Returns functions supported by a driver. This function allows an application to determine at run-time whether a particular function is supported by a driver.
SQLGetInfo	Retrieves general information about a driver and data source, including file names, versions, and the maximum length of names supported by the data source.

Function Name	Description
SQLGetTypeInfo	Determines the data types supported by a driver and data source.
SQLSetStmtOption **SQLSetConnectOption** **SQLGetStmtOption** **SQLGetConnectOption**	Set or retrieve operational parameters for the driver and data source. Options include access mode, timeout values, implicit commit operation, and asynchronous execution of ODBC functions.

SQLDriverConnect

XE "SQLConnect:compared to SQLDriverConnect"§XE "SQLDriverConnect"§The **SQLDriverConnect** function allows you to request that the driver and Driver Manager obtain login information from the user prior to establishing a connection. **SQLDriverConnect** uses a connection string to connect to a driver and data source. This function is useful if the data source requires information that cannot be supplied in the standard **SQLConnect** function.

A connection string contains the following information:

- Data source name
- One or more user IDs
- One or more passwords
- One or more database-specific parameter values

The connection string is a more flexible interface than the data source name, user ID, and password used by **SQLConnect**. You can use the connection string for multiple levels of logon authorization or to convey other data source-specific connection information.

You can call **SQLDriverConnect** in two ways:

- Specify a connection string that allows the driver to connect to the data source.
- Specify a partial connection or no connection string. The Driver Manager displays a dialog that allows the user to select a data source name. The driver displays a login dialog, includes any partial information as default values, obtains necessary connection information from the user, and then establishes the connection.

Once the driver establishes a connection, **SQLDriverConnect** returns a connection string that you can use to call **SQLDriverConnect** again later, if

necessary.

The Driver Manager displays the following dialog if the application calls **SQLDriverConnect** and requests that the user be prompted for information.

Upon request from the application, the driver displays a dialog similar to the following to retrieve login information.

6 Preparing and Executing an SQL Statement

XE "SQL statements:preparing and executing"§Your application can submit the SQL statements listed in Appendix A, “SQL Grammar” (or data source-specific SQL statements) in ODBC function calls. The list in Appendix A is similar to the set of SQL statements that can be prepared in embedded SQL.

§

Note To request a commit or rollback operation, call **SQLTransact**.

§

The following diagram shows a sample sequence of ODBC commands that can be used for SQL statement processing. For more information about statement sequencing, refer to the *ODBC API Reference*, Appendix B, "ODBC State Transition Table."

Note that any valid SQL statement can be executed with either the **SQLPrepare** and **SQLExecute** sequence or the **SQLExecDirect** command, depending on whether the you plan to submit the SQL statement once or more than once. This functionality differs from embedded SQL, since both statements with and without cursors are executed the same way.

Note also that there are other valid calling sequences that include functions such as **SQLBindCol** and **SQLGetData**.

Sample Flow Control

This chapter describes ODBC functions that support SQL statement processing. For information about results, refer to Chapter 7, “Retrieving Results.”

Allocating a Statement Handle

XE "Statement handles:requesting"§XE "Handles:requesting statement"§Allocate a statement handle as follows:

1. Declare a variable of type **HSTMT**. For example, you could declare “HSTMT hstmt1.” **HSTMT** is defined in the SQL.H file, listed in the *ODBC API Reference*, Appendix F, “ C Header Files.” For Windows applications, the hstmt variable is a memory handle.
2. Pass the address of this variable and an existing connection handle in a call to **SQLAllocStmt**.

The driver allocates a statement handle and returns the handle to your application.

Assigning Storage for Results (Binding)

XE "Binding result columns"§XE "Results:binding"§XE "Results:assigning storage for"§You can assign storage for results before or after you execute an SQL statement. XE "SQLBindCol"§To allocate storage for a column of data, call **SQLBindCol** and include the following information:

- Decide whether you want the driver to convert the results to a different data type. If so, include this request in your call to **SQLBindCol**.
- Define a storage buffer for the data. This storage area must be large enough to hold the maximum number of bytes needed for the column for the specified data type.
- Define a storage buffer for the data length.

Choosing Prepared or Direct Execution

XE "Prepared execution"§XE "Direct execution"§XE "SQL statements:executing"§You have two execution options when you send an SQL request:

- Prepared Use this option if you want to execute the same statement more than once without respecifying the SQL string or if you need information about the result set prior to execution.

- Direct Use this option if you want to submit the statement once and you do not need the result format prior to execution.

These two options differ from the prepared and immediate options in embedded SQL. For a comparison between ODBC functions and embedded SQL, refer to the *ODBC API Reference*, Appendix E, "Comparison Between Embedded SQL and ODBC."

Prepared Execution

If you plan to submit the SQL statement multiple times, possibly with intermediate changes to parameter values, prepare the request as follows.

XE "SQLPrepare"§XE "SQLSetParam"§XE "UPDATE (SQL statement):positioned"§XE "DELETE (SQL statement):positioned"§XE "SQL statements:positioned UPDATE and DELETE"§XE "SQLGetCursorName"§XE "SQLSetCursorName"§Issue the following two calls in any order after **SQLAllocStmt** and prior to **SQLExecute**:

- Call **SQLPrepare** and pass the SQL statement as an argument.
- If your SQL statement includes parameter markers, call **SQLSetParam** to associate storage areas with corresponding parameter markers. If this is not the first execution of an SQL statement, you can reuse previous storage areas.

If your request requires a cursor name, as in positioned update or delete (**UPDATE WHERE CURRENT OF** *cursor-name* or **DELETE WHERE CURRENT OF** *cursor-name*), you can allow the driver to generate the cursor name or you can call **SQLSetCursorName** to associate a cursor name with your prepared request.

After setting the cursor name with **SQLSetCursorName** or implicitly obtaining a cursor name by executing a **SELECT** statement, you can call **SQLGetCursorName** to retrieve the cursor name. (The following subsection, “Performing Positioned Updates and Deletes,” contains additional information about positioned updates and deletes.)

Set parameter values for all parameter markers, and then call **SQLExecute** to submit the request.

XE "Declare Cursor (SQL statement)"§If you do not set new parameter values prior to a subsequent call to **SQLExecute**, the driver reuses existing parameter values.

XE "Prepared SQL statements, advantages"§The prepare and execute approach provides the following advantages:

- If the data source supports statement preparation, this is the most efficient way to perform multiple iterations of the same request, especially for complex SQL statements. The data source minimizes processing time by compiling the SQL statements once, producing an access plan, then using the plan for each execution of the request. An access plan identifier allows the driver to send a tag instead of the full SQL statement for subsequent requests, thus minimizing network traffic on subsequent executions of a statement.
- You can retrieve information about the format of the result set prior to executing the SQL statement.

Direct Execution

XE "SQLExecDirect"§If you do not require information about the result set prior to completion of your SQL request and you plan to submit a statement only once, you can call **SQLExecDirect** to submit the SQL statement.

Processing Positioned Updates and Deletes

A positioned update or positioned delete performs an update or delete operation, respectively, based on cursor position.

XE "UPDATE, positioned"§XE "DELETE, positioned"§XE "SQL statements:positioned UPDATE and DELETE"§After you submit a **SELECT** statement that returns multiple rows and you fetch one or more result rows, you can perform a positioned **UPDATE** or **DELETE** to update or delete the row

referenced by the cursor. To request a positioned update or delete operation, use the following SQL syntax:

UPDATE {tablename | viewname} SET {columnname = expression} WHERE CURRENT OF cursorname

DELETE FROM {tablename | viewname } WHERE CURRENT OF cursorname

Next, execute the statement:

- Prepared execution Call **SQLPrepare** with szSqlStr set to the text of the positioned **UPDATE** or **DELETE** statement. Use a different hstmt than that used for the **SELECT** statement. Include the cursor name associated with the earlier **SELECT** statement. Set parameter values as necessary, and then call **SQLExecute** to submit the statement.
- Direct execution Call **SQLExecDirect** with szSqlStr set to the text of the **UPDATE** or **DELETE** statement. Use a different hstmt than that used for the **SELECT** statement. Include the cursor name associated with the earlier **SELECT** statement.

The driver associates the new hstmt with the existing hstmt , includes parameter values if necessary, submits the positioned **szSqlStr** or **DELETE** statement, and returns results to the application.

Example

The following diagram lists sample calling sequences for prepared and direct positioned update or delete operations. This sequence is an example; you could include calls to **SQLSetParam** in either sequence or combine prepared and direct requests in the same processing stream.

Extensions for Processing SQL Statements

The following table lists related extended functions. The paragraphs following this table describe the following topics in more detail: retrieving data dictionary information; sending arrays of parameters; sending large data values; using scrollable cursors; requesting asynchronous processing; and requesting scalar functions, extended data types, and outer joins.

To determine whether a particular driver supports these functions, call **SQLGetFunctions**.

Function or Operation	Description
Data dictionary functions	Return data dictionary information. These functions are useful if the data source does not support SQL system views.
SQLDescribeParam	Return information about prepared parameters.
SQLGetData	Return one column of one row of data to the application. **SQLGetData** is useful for returning large data values.
Sending large data values	Allow the application to send large data values to the data source.
SQLParamOptions	Allow the application to specify multiple sets of parameter values for a single SQL statement. This capability, if supported by the data source, minimizes network traffic..
SQLSetScrollOptions	Establish a scrollable cursor for the result set.
Asynchronous processing	Allow the application to request asynchronous processing of a subset of ODBC functions.

Scalar functions	Support the use of scalar functions in SQL statements.
Extended data types	Support embedded extended data type literals in SQL statements.
Outer joins	Support outer join requests in SQL statements.

Obtaining Information about the Data

The following functions return information about data:

- **SQLColumnPrivileges** returns a list of columns and associated privileges for one or more tables
- **SQLColumns** returns the list of columns names in a specified table
- **SQLForeignKeys** returns a list of column names that compose foreign keys for a specified table
- **SQLPrimaryKeys** returns the column name (or names) that comprise the primary key for a table
- **SQLSpecialColumns** returns information about the optimal set of columns that uniquely identifies a row in a table or the columns that are automatically updated when any value in the row is updated by a transaction
- **SQLStatistics** returns a list of statistics about a single table and the indexes associated with the table
- **SQLTablePrivileges** returns privileges associated with one or more tables
- **SQLTables** returns the list of table names stored in a specific data source

Each function returns the information as a result set. An application fetches these results in the same manner as it retrieves query results (through a call to **SQLFetch**).

Sending Large Data Values

XE "Large data values:sending"§To send large data values, use the following three functions:

- **SQLSetParam**

- **SQLParamData**
- **SQLPutData**
- **SQLParamOptions**

To indicate that you plan to send a large data value, call **SQLSetParam** to associate storage with the parameter—and set pcbValue to SQL_LONG_DATA for the parameter.

In the call to **SQLSetParam**, set rgbValue to a value that, at run time, references the location of the data. The driver returns this value to the application at statement execution time.

When the driver processes a call to **SQLExecute** or **SQLExecDirect**, the driver returns SQL_NEED_DATA as soon as it encounters a parameter that requires a large data value. The application then calls **SQLParamData** and **SQLPutData** to send data values:

- **SQLParamData** searches for the next large data value parameter and returns the value referenced by rgbValue (in the earlier call to **SQLSetParam**).
- **SQLPutData** transports the actual data value to the data source.

For additional information, refer to the description of **SQLSetParam** in Chapter 2 of the *ODBC API Reference*.

Specifying Arrays of Parameter Values

XE "Parameters:specifying arrays of"§XE "SQLParamOptions"§To specify multiple sets of parameter values for a single SQL statement, call **SQLParamOptions**. For example, if you have ten sets of column values to insert into a table—and you can use the same SQL statement for all ten operations—you can set up an array of values, then submit a single INSERT statement.

If you use **SQLParamOptions**, your application must allocate enough memory to handle the arrays of values.

Using Scrollable Cursors

SQL was originally designed to return one row at a time to an application. Scrollable cursors provide more flexible access to blocks of result data. XE "Scrollable cursors, overview"§The following paragraphs provide an overview of scrollable cursors and describe ODBC features that support scrollable cursors.

Basic Cursors

An SQL **SELECT** statement extracts data that meets a set of specifications. For example, SELECT * FROM EMPLOYEE WHERE EMPNAME = "JONES" returns all columns of all rows in EMPLOYEE where the employee's name is Jones. This set of information, called a result set, can contain zero, one, or more than one row.

Applications retrieve single rows as follows:

- A cursor, managed by the driver, points to the current row in the result set.
- A call to **SQLFetch** moves the cursor to the next row in the result set and retrieves the row.

This basic form of a cursor is called a forward-only scrolling cursor, and is supported by core ODBC functions. To fetch a previous row using a forward-only cursor, the driver closes the cursor, reopens the cursor for the same result set, and fetches rows until it retrieves the target row.

Scrollable Cursors

Scrollable cursors allow a user to scan results in a flexible manner without excessive support from the application. Users can view rows within a block of data and update, delete, refresh, or browse through the data. Scrollable cursors use the following concepts:

- A block of data is called a rowset.
- A set of keys that uniquely identifies the rows in a rowset is called a keyset. If a table does not contain unique key fields, the keyset may be the whole row.

XE "Concurrency control, setting"§As the size of a rowset increases, so does the possibility that another user may want to access or update one of the rows. You can request four types of locking for a keyset:

- Read only Read the data and build a keyset, but do not lock the data. This approach does not guarantee that the key will point to the same row at a later time.
- Locked Read the data with a lock. Other users cannot modify the data until you remove the lock. This approach guarantees that data is the same when a subsequent update or delete is performed.
- Optimistic concurrency control comparing timestamps Do not lock the data.

Instead, store the time the row was last modified (if available). If the user requests a positioned update or delete operation, check the timestamps to make sure the row was not modified since the keyset was built.

- Optimistic concurrency control comparing values Do not lock the data. Instead, include all row data values in the keyset. If the user requests a positioned update or delete operation, compare these values to values in the database to make sure the row was not modified since the keyset was built.

Call **SQLSetScrollOptions** to specify rowset, keyset, and concurrency control. If the application uses scrollable cursors, the application must call **SQLSetScrollOptions** before it calls **SQLPrepare** or **SQLExecDirect**.

SQLBindCol binds storage areas for result columns.

XE "SQLExtendedFetch"§To fetch a block of data, call **SQLExtendedFetch**. Following the extended fetch, the cursor points to the entire rowset for subsequent positioned operations. To position the cursor on specific rows within the rowset, call **SQLSetPos**.

For scrollable cursor operations, the rowset behaves as a single fat cursor.XE "SQLSetPos"§

Regardless of position, a fetch next or fetch previous operation moves the entire rowset as if it were one cursor.

The **SQLExtendedFetch** function supports forward, backwards, and arbitrary retrieval of blocks, so that there are two levels of movement within a result set: at the rowset level and at the row level.

Requesting Asynchronous Processing

XE "SQL statements:processing asynchronously"§By default, a driver processes ODBC functions synchronously; the driver does not return control to your application until a function call completes. If a driver supports asynchronous processing, however, you can request asynchronous processing for the functions listed below.

SQLColAttributes	**SQLExecDirect**	**SQLGetTypeInfo**	**SQLSetPos**
SQLColumns	**SQLExecute**	**SQLMoreResults**	**SQLSpecialColumns**
SQLColumnPrivileges	**SQLExtendedFetch**	**SQLNumResultCols**	**SQLStatistics**

SQLConnect	**SQLFetch**	**SQLParamData**	**SQLTablePrivileges**
SQLDescribeCol	**SQLForeignKeys**	**SQLPrepare**	**SQLTables**
SQLDescribeParam	**SQLGetData**	**SQLPrimaryKeys**	
SQLDriverConnect	**SQLGetInfo**	**SQLPutData**	

All of these functions either submit requests to a data source or retrieve data. Any of these functions can initiate extensive processing.

To enable or disable asynchronous processing for the above set of functions, call **SQLSetStmtOption** and specify ON or OFF for the SQL_ASYNC_ENABLE option. To check the setting of the SQL_ASYNC_ENABLE option, call **SQLGetStmtOption**. To enable or disable asynchronous processing for all hstmts associated with an hdbc, call **SQLSetConnectOption**.

Upon successful initiation of an asynchronously-initiated function, the driver returns SQL_STILL_EXECUTING. If you resubmit the function call, the driver returns SQL_STILL_EXECUTING until the function completes. Once the function completes, the driver returns a standard return code.

Using Extended Data Types, Functions, and Outer Joins

ODBC supports an escape mechanism that allows an application to embed database-specific data types, scalar functions, or outer join specifications within an SQL statement. XE "Escape sequence:data types"§The escape mechanism follows the format for vendor-specific commands as defined in Appendix A, "SQL Grammar." In addition, ODBC defines canonical forms for the following:

- Date, time, and timestamp data types
- Scalar functions such as string and numeric functions
- Data type conversion

- Outer join request

The application can use one of two forms when specifying a data type or scalar function:

- The canonical form as defined by ODBC. This approach provides database independence. The driver translates the canonical form into the database-specific form.
- The form required by the data source. This approach does not provide database independence.

§

Note An application can submit literal data, without using an escape sequence, as long as it calls **SQLGetTypeInfo** to make sure the data type is supported and to obtain appropriate prefix and suffix information.

§

The following paragraphs list the syntax for each type of escape clause. In addition, ODBC defines a shorthand syntax for escape clauses, defined in Appendix A, "SQL Grammar."

Date and Time Data Types

The following escape clause allows an application to specify a date, time, or timestamp data type:

--*(vendor(Microsoft),product(ODBC), {d|t|ts} *value* **--*)**

The following table describes each element of the preceding clause.

Argument	Description
vendor(Microsoft), product(ODBC)	Identifies the vendor and product that support the escape clause. Vendor and product are not used with the shorthand notation.
d	Indicates value is in date format (yyyy-mm-dd).
t	Indicates value is in time format

	(hh:mm:ss).
ts	Indicates value is in timestamp format (yyyy-mm-dd hh:mm:ss.[ffffff]), where the precision represented by fraction of a second ([ffffff]) depends on the data source).
value	The value of the date, time, or timestamp variable.

You can specify a date, time, or timestamp escape clause in place of a literal in an SQL statement. For detailed syntax information, refer to Appendix A, "SQL Grammar."

To submit a data type in the native form of the data source, submit the SQL statement with vendor-specific syntax.

Call **SQLGetFunctions** to determine if a driver supports a specific extended data type.

For a list of core and extended data types, refer to the *ODBC API Reference*, Appendix D, "Data Type Definitions."

Scalar Functions

Scalar functions—such as string length, absolute value, or current date—can be used on columns of a result set and on columns that restrict rows of a result set. ODBC supports a set of canonical scalar functions that may be a subset or superset of functions actually supported by a given DBMS. If the application specifies the canonical form of a scalar function, the driver translates the function to the syntax required by the data source. If the application specifies the native form of a scalar function, the driver does not translate the function, but sends it to the data source in the form specified by the application. In either case, the data source makes the final determination of the validity of the scalar function and its arguments.

The following escape clause allows an application to specify a scalar function:

--*(vendor(Microsoft),product(ODBC), fn *function*--*)

The following table describes each element of the preceding clause.

Argument	Description
vendor(Microsoft), product(ODBC)	Identifies the vendor and product that support the escape clause. Vendor and product are not used with the shorthand notation.
fn	Indicates that the escape sequence requests a scalar function.
function	Is an expression that contains one or more function names and function arguments. For a list of canonical functions, refer to the *ODBC API Reference*, Appendix G, "Canonical Functions."

You can use an escape clause with a canonical function in place of a scalar function in an SQL statement. For information about the syntax of canonical functions, refer to Appendix A, "SQL Grammar," and the *ODBC API Reference*, Appendix G, "Canonical Functions."

An application can also include native DBMS functions in an escape clause. The following example shows how an application would specify a native function ("round," in this example) and two ODBC canonical functions (ABS and SQRT). This example uses shorthand escape clause syntax, as defined in Appendix A,

"SQL Grammar," and accesses three columns—EMPNO, EMPNAME, and EMPDIST—in a table called EMPLOYEE.

```
SELECT EMPNO, {fn abs( round ( {fn sqrt(EMPNAME)} ))}, EMPDIST
FROM EMPLOYEE
```

An application can call **SQLGetInfo** to determine which functions are supported by a specific driver and associated data source.

Data Type Conversion Function

ODBC defines a special type of canonical function, called the CONVERT function, that allows applications to explicitly request a data type conversion when the database processes the SQL statement. The driver translates the canonical form into the database-specific form.

The canonical form of the CONVERT function does not restrict the range of data type conversions. Instead, each driver determines the valid set of conversions. Call **SQLGetInfo** to determine which conversions are supported by the data source and its associated driver.

The following example shows how an application would specify a conversion to a character data type:

```
SELECT EMPNO FROM EMPLOYEE WHERE
--
*(vendor(Microsoft),product(ODBC),fn(CONVERT(EMPNO,SQL_CHAR))--
*) LIKE '1%'
```

For more information, refer to the *ODBC API Reference*, Appendix G, "Canonical Functions."

Requesting Outer Joins

XE "Escape sequence:outer joins"§The ODBC interface uses an escape sequence to support outer joins. The format is:

```
--*(vendor(Microsoft),product(ODBC) oj join-type --*)
```

Join-type specifies a type of outer join. For example, to perform an outer join between two tables named EMPLOYEE and DEPT, using the DEPT.ID column for the join condition, use the following statement (shown in shorthand escape clause syntax):

```
SELECT employee.name, dept.name FROM
 {oj employee LEFT OUTER JOIN dept ON employee.deptid=dept.deptid}
 WHERE employee.projid=544
```

This syntax is a subset of ANSI SQL2 outer join syntax.

If you use an escape clause with a outer join request in an SQL statement, the escape clause must appear after the FROM clause and before the WHERE clause, if either exist in the statement. For detailed syntax information, refer to Appendix A, "SQL Grammar."

Call **SQLGetFunctions** to determine if a driver supports a specific join type and, if so, what join characters to use in *value*.

7 Retrieving Results

XE "Results:processing"§This chapter describes result-handling operations.

The steps you take to process results depend on your knowledge about the result set.

- Known result set You know the exact form of your SQL statement prior to execution. For example, the query **SELECT EMPNO, EMPNAME FROM EMPLOYEE** returns two specific columns. The statement **DELETE FROM EMPLOYEE WHERE EMPNAME = "Harry Jones"** returns a row count.
- Variably structured result set You do not know the exact form of your results at compile time. For example, the ad-hoc query **SELECT * FROM EMPLOYEE** returns all currently defined columns in the EMPLOYEE table. You may not be able to predict the format of these results prior to execution.

Determining Characteristics of a Result Set

XE "SQLNumResultCols:determining result characteristics"§XE "SQLRowCount:determining result characteristics"§XE "SQLDescribeCol:determining result characteristics"§XE "Results:determining characteristics"§To determine the characteristics of a result set, perform these steps:

1 Call **SQLNumResultCols** to see if your request returned a set of rows.

2 If **SQLNumResultCols** returns a nonzero result, your request was a **SELECT** statement. Call **SQLDescribeCol** and **SQLBindCol** for each column, and then call **SQLFetch** to retrieve each row.

   If **SQLNumResultCols** returns a zero, call **SQLRowCount** to see if your request inserted, deleted, or modified rows:

   - If **SQLRowCount** returns a nonzero result, that result is the number of rows modified.
   - If **SQLRowCount** returns a zero, the statement performed other work: commit, rollback, or other operation that does not modify or return rows.

§

Note If you prefer to check for row modification first, call **SQLRowCount** before you call **SQLNumResultCols** .

§

Fetching Result Data

XE "Results:retrieving data"§If you requested an SQL statement that does not manipulate data (for example, **GRANT** or **REVOKE**), the return code for the ODBC function call indicates whether your request was successful or not.

If you requested an **INSERT**, **UPDATE**, or **DELETE** operation, **SQLRowCount** returns the number of rows affected by the operation.

XE "SQL statements:processing results"§XE "SELECT (SQL statement):storing results"§XE "SQLFetch:retrieving result rows"§If you requested a **SELECT** statement and you did not bind the columns to storage locations prior to execution, call **SQLBindCol** to perform bind operations prior to retrieving results. Call **SQLFetch** to retrieve each row of results.

The following diagram shows the flow of the preceding two operations:

Retrieving Error and Status Information

XE "SQLError"§XE "Errors, retrieving"§If your ODBC call returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, call **SQLError** to find out more information. **SQLError** returns standardized messages and data source-specific messages. The application is responsible for reading these messages and taking appropriate action.

For information about how to handle standard error and status messages, refer to “Calling ODBC Functions” in Chapter 3. For more information about ODBC error codes, refer to the *ODBC API Reference,* Appendix A, “ODBC Error Codes.”

Related Extension Functions

XE "SQLNativeSql"§XE "SQLDescribeParam"§XE "Escape sequence:translating to SQL text"§XE "Parameters:obtaining data type information"§The following table lists related extended functions. The paragraphs following this table contains additional information about **SQLGetData**, **SQLExtendedFetch**, and retrieving multiple result sets.

Function Name	Description
SQLNativeSql	Retrieve the SQL statement as processed by the data source, with escape sequences translated to native SQL code.
SQLDescribeParam	Retrieve data type information for parameters in an SQL statement.
SQLParamOptions	Specify multiple values for a set of parameters.
SQLGetData	Fetch one column in a result row. This function is useful for retrieving large data values.
SQLExtendedFetch	Fetch a block of result data.
SQLSetPos	Position the cursor within a fetched block of data.
SQLScrollOptions	Request a scrollable cursor.

Retrieving Data in Large Columns

To retrieve data from large columns—or retrieve data one column at a time—call

SQLGetData to retrieve the column into a buffer. If you use **SQLGetData** to retrieve data, do not call **SQLBindCol** for the column.

SQLFetch does not retrieve data for columns that are not bound, but does move the cursor from row to row and retrieve data in bound columns. Prior to calling **SQLGetData**, call **SQLFetch** to position the cursor at the first row. Call **SQLFetch** to move the cursor to subsequent rows.

You can combine **SQLGetData** with **SQLBindCol** and **SQLFetch** access within a row. To combine these operations, bind as many contiguous columns as necessary. Call **SQLFetch** to retrieve all bound rows and leave the pointer at the end of the bound rows. Call **SQLGetData** to retrieve the unbound columns. (Note, however, that you cannot use **SQLGetData** to retrieve a column to the left of the rightmost bound column.)

You can call **SQLGetData** multiple times for a single column, if necessary.

Processing Multiple Result Sets

Call **SQLMoreResults** function to process multiple result sets. XE "Stored procedures, processing results"§XE "Batched SQL statements, results"§XE "SQL statements:batched processing"§The following SQL statements can return multiple result sets:

- Batched SQL statements You submit more than one **SELECT** statement in one request.
- Statements with arrays of parameters You submit more than one set of parameter values for a SQL statement (**SQLParamOptions**).
- Stored procedures You submit a stored procedure name as an SQL statement; such a procedure can return multiple result sets.

Processing Blocks of Results

If the driver supports scrollable cursors, your application can retrieve blocks of result rows. For more information about scrollable cursors and processing blocks of result data, refer to "Using Scrollable Cursors," in Chapter 6.

8 Terminating Transactions and Connections

XE "Transactions:terminating"§XE "Handles:releasing statement"§XE "Statement handles:releasing"§XE "Statements, terminating"§The ODBC interface provides functions that terminate SQL transactions, statement-processing connections (hstmts), connections (hdbcs), and environment connections (henvs).

Terminating Statement Processing

Call **SQLFreeStmt** to free resources associated with a statement handle. The **SQLFreeStmt** function has four options:

- **SQL_CLOSE** Close the cursor, if one exists, and discard pending results. You can use the statement handle again later.
- **SQL_DROP** Close the cursor, if one exists, discard pending results, and free all resources associated with the statement handle.
- **SQL_UNBIND** Release all return buffers bound by **SQLBindCol** for the statement handle.
- **SQL_RESET_PARAMS** Release all parameter buffers requested by **SQLSetParam** for the statement handle.

Terminating Transactions

XE "SQLTransact:terminating a transaction"§At the transaction level, call **SQLTransact** to commit or roll back the current transaction.

XE "SQLCancel:terminating statement processing"§XE "SQLError:checking statement cancellation"§To cancel a specific statement, call **SQLCancel**, check the return code of the function that was canceled, and then call **SQLError** to determine the status of the cancellation. For additional information about **SQLCancel**, refer to the *ODBC API Reference,* Chapter 2, “ODBC API Reference.”

Terminating Connections

XE "Connections:terminating"§XE "SQLFreeConnect:releasing a connection handle"§XE "SQLDisconnect:closing a connection"§To terminate a connection to a driver and data source, perform the following steps:

1. Call **SQLDisconnect** to close the connection. You can then use the handle to reconnect to the same data source or to a different data source.

2. Call **SQLFreeConnect** to release the connection handle and free all resources associated with the handle.
3. Call **SQLFreeEnv** to release the environment handle and free all resources associated with the handle.

9 Constructing an ODBC Application

XE "Developer's Kit, contents"§The following paragraphs describe how to construct an application, starting with the contents of the developer's kit and including sample code, setup information, and references to testing and debugging information. This manual assumes that you are developing an application for the Windows environment.

Developer's Kit Contents

Your development kit should contain the following items:

- SDK cover letter and license agreement
- SDK installation instructions
- *ODBC Application Programmer's Guide* (this manual), *ODBC Driver Developer's Guide*, *ODBC API Reference, ODBC Test Application User's Guide*, a binder, and tabbed dividers
- One 3.5-inch, 1.44 MB disk and one 5.25-inch, 1.2 MB disk, both containing:
  - Installation batch file and utilities (including INSTALL.BAT, INSTALL2.BAT, and CHKDIR.EXE)
  - Driver Manager (DRVRMGR.DLL)
  - Driver Manager Import Library (DRVRMGR.LIB)
  - Test application (GATOR.EXE)
  - Sample Driver DLL (SAMPLE.DLL)
  - ODBC core functions header file (SQL.H)
  - ODBC extension functions header file (SQLEXT.H)
  - ODBC initialization file (ODBC.INI)
  - Test application initialization file (GATOR.INI)
  - Common dialogs DLL (COMMDLG.DLL)
  - Test application source files, header files, object files, makefile, and other files
  - Sample driver source files, header files, make file, and other files
  - Other common header files
  - Other common libraries

System Requirements

The following paragraphs list hardware, software, and environmental requirements for the ODBC environment.

Hardware Requirements

ODBC requires approximately two megabytes of disk space for installation of SDK files and for assembling, compiling and linking the test application and sample driver.

The SDK software has been tested on the following hardware, although it may be possible to use other configurations:

- Personal computer with an 80386 processor and at least five megabytes of RAM.

Software Requirements

The SDK software has been tested with the following system and development software, although it may be possible to use other configurations:

- MS-DOS 5.0
- Microsoft Windows 3.0 or Windows 3.1 Beta-3
- Microsoft C6.00A
- Microsoft Windows 3.0 SDK or Windows 3.1 Beta-3 SDK
- Microsoft MASM 5.3

Environmental Requirements

When using the supplied makefiles to build customized versions of the test application (gator.exe) or the sample driver (sample.dll), you must be sure that the \INCLUDE and \LIB directories in your ODBC SDK directory are specified in your PATH or environment variables as specified in the *Microsoft C Compiler Reference Manual.* These two directories should be searched first during the compile and link process. You must also make sure that the source, object and other files for the test application or sample driver can be found by the assembler, compiler and linker. Review the makefiles and any necessary assembler, compiler and linker documentation to ensure that you have your environment correctly defined for your configuration.

Installing the Developer's Kit

XE "Installing ODBC"§XE "ODBC:installing"§To install the software for the SDK, refer to the SDK installation instructions.

Upon successful completion of the installation process, your Windows directory should contain the ODBC initialization file (ODBC.INI). The directory that you specified for the installation of the rest of the SDK files should contain the following files (the default ODBC directory will be used for purposes of illustration):

Directory	File	File Description
C:\WINDOWS	ODBC.INI	The ODBC initialization file. Includes a data source specification for the SAMPLE.DLL and the sample driver.
C:\ODBC	GATOR.EXE	ODBC GATOR; driver test application.
	DRVRMGR.DLL	ODBC Driver Manager DLL.
	COMMDLG.DLL	Common Dialogs DLL.
	SAMPLE.DLL	ODBC Sample Driver DLL.
	GATOR.INI	ODBC GATOR initialization file.
C:\ODBC\GATOR	AUTOTEST.C	Source for selecting and initiating auto-tests. For example, CORQTEST.C is the source for an autotest and is initiated from here. Custom autotests can be hooked into GATOR via this file.
	AUTOTEST.H	Header file for AUTOTEST.C

	CORQTEST.C	Source for the core conformance quick test autotest.
	CUSTOM.C	An autotest prototype that can be used as the basis for creating customized auto-tests.
	GATOR.DEF	Windows module definition file for GATOR.
	GATOR.H	GATOR header file.
	GATOR.RES	Windows resource file for GATOR.
	L1QTEST.C	Source for the ODBC conformance level 1 quick test autotest.
	L2QTEST.C	Source for the ODBC conformance level 2 quick test autotest.
	MAKEFILE	GATOR makefile; a working model.
	SQL_0001.ACT	Results from the query defined by the file SQL_0001.QRY.
	SQL_0001.QRY	Example of a query file that can be run against the sample driver.

Directory	File	File Description
C:\ODBC\GATOR (continued)	SQL_0001.RST	The comparison file for the results of SQL_0001.QRY.

	TESTS.H	Contains references to global variables used in GATOR.
C:\ODBC\GATOR\OBJ	AUTO.OBJ	The object file that supports the "Auto" menu selections in GATOR.
	CONNECT.OBJ	The object file that supports the "Connect" menu selection in GATOR.
	DDA.OBJ	The object file that supports the "Dictionary" menu selection in GATOR.
	GATOR.OBJ	The object file for the basic GATOR application.
	MFQENG.OBJ	The object file that supports the execution of query files (.QRY) from the "Auto-Tests..." menu selection in GATOR.
	MISC.OBJ	The object file that supports the "Misc" and "Options" menus selections in GATOR.
	RECEIVE.OBJ	The object file that supports the "Results" menu selections in GATOR.
	SEND.OBJ	The object file that supports the "Statements" menu selections in GATOR.
	SERVERCN.OBJ	The object file that supports the "tools" section of the the "Connect" menu in GATOR.

C:\ODBC\INCLUDE	SQL.H	ODBC core functions header file.
	SQLEXT.H	ODBC extension functions header file.
	WINDOWS.H	Microsoft Windows header file.
C:\ODBC\LIB	COMMDLG.LIB	Common dialogs import library.
	DRVRMGR.LIB	ODBC Driver Manager import library.
	LIBW.LIB	Microsoft Windows library.
	MDLLCEW.LIB	Microsoft Windows library.
	MLIBCEW.LIB	Microsoft Windows library.
C:\ODBC\SAMPLE	CONNECT.C	Sample driver implementation of: **SQLAllocEnv**, **SQLFreeEnv**, **SQLAllocConnect**, **SQLConnect**, **SQLDisconnect** and **SQLFreeConnect**.
	DATABASE.C	Sample driver "database" functions..
	DATABASE.H	Header file for DATABASE.C.

Directory	File	File Description
C:\ODBC\SAMPLE	ECONNECT.C	Sample driver implementation of:

(continued)		**SQLDriverConnect** and **SQLGetInfo**.
	EDATA.C	Sample driver implementation of: **SQLGetTypeInfo**.
	EDICT.C	Sample driver implementation of: **SQLTables** and **SQLColumns**.
	EMISC.C	Sample driver implementation of: **SQLSetConnectOption**, **SQLSetStmtOption**, **SQLGetConnectOption** and **SQLGetStmtOption**.
	ERESULTS.C	Sample driver implementation of: **SQLGetData**
	ERR.H	Header file mapping error code constants for SQLSTATE values to error values.
	EXECUTE.C	Sample driver implementation of: **SQLAllocStmt**, **SQLPrepare**, **SQLSetParam**, **SQLSetCursorName**, **SQLGetCursorName**, **SQLExecute**, and **SQLExecDirect**.
	LIBSTART.ASM	Entry point startup routine for Windows sharable code libraries.
	MAKEFILE	Sample driver makefile; a working model.
	MEMORY.C	Memory and resource management cover functions.

MISC.C	Sample driver implementation of: **SQLError** and **SQLCancel**.
RESULTS.C	Sample driver implementation of: **SQLNumResultsCols**, **SQLDescribeCol**, **SQLBindCol**, **SQLFetch**, **SQLRowCount** and **SQLFreeStmt**
SAMPLE.DEF	Windows module definition file for the sample driver.
SAMPLE.H	Overall header file for the sample driver.
SAMPLE.RC	Windows resource file for the sample driver.
TRANSACT.C	Sample driver implementation of: **SQLTransact**.
WEP.C	Necessary for writing DLLs for Windows 3.0.

Directory	File	File Description
C: \ODBC\SAMPLE\DEB UG		The directory where object files for the sample driver are placed at compile time, if DEBUG is set as an environment variable.
C:\ODBC\SAMPLE \NODEBUG		The directory where object files for the sample driver are placed at compile time, if DEBUG is not set as an environment variable or if it is set to null.

Several components of the Windows SDK have been included with the ODBC SDK. These need only be used if you are using a beta version of the Windows 3.1 SDK. If you are using a beta version of the Windows 3.1 SDK, make sure that the \INCLUDE and \LIB directories of your ODBC SDK are searched first during compilation and link. The specific files of interest are WINDOWS.H, LIBW.LIB, MDLLCEW.LIB, MLIBCEW.LIB, COMMDLG.LIB and the COMMDLG.DLL.

Constructing an ODBC Environment

The ODBC environment uses an initialization file, called ODBC.INI, to store data source names and related information.

The ODBC.INI file stores information used by the ODBC Driver Manager, ODBC drivers and the ODBC SETUP routine. For example, all connection-related ODBC functions (**SQLConnect**, **SQLDataSources**, and **SQLDriverConnect**) accept a data source name (DSN) as an argument or as an element of an argument.

ODBC drivers can read the ODBC.INI and can update it in certain instances. The ODBC Driver Manager reads the ODBC.INI file, but does not update it. Applications should not read directly from ODBC.INI. ODBC functions supply information from the initialization file in a consistent and structured manner.

There are two types of sections in the ODBC.INI file. One type of section defines the data sources accessible through ODBC. These are called data source specifications. The ODBC.INI file can contain one or more data source specifications. The section name of each data source specification defines the data source name associated with the specification.

The other type of section is a list of the data source names of each of the data source specifications. This is a single section that contains all valid data source names.

Data Source Specification

Before accessing a driver or data source, the data source must be defined in the ODBC initialization file. Each data source definition resides in a separate section in the ODBC.INI file. The data source definition section is called a data source specification.

A data source specification, at a minimum, consists of:

- A data source name (this must be the name that appears in the section heading)
- The name of an ODBC driver
- A description of the data source

The data source name and description are defined by the user.

The following depicts a basic data source specification:

[data-source-name]
driver = *driver-DLL-name*
description = description-of-data-source

The specification can include driver-specific information, as well. Driver-specific information can be supplied by the user or by the driver. Some examples of driver-specification information are:

server=<server-name>

lastuid=<last user id used to logon>
database=<database name>
OemAnsi=<conversion indicator>

The ODBC SETUP routine allows users to add driver-specific information to the ODBC initialization file.

Default Data Source Specification

The ODBC.INI file can contain a single default data source specification. This default specification is optional. If the default specification exists, the data source name must be "default" and the driver can be any one of the set of installed drivers. When initially defined through ODBC.SETUP, the default data source specification consists of only the data source name and the driver DLL. The driver can add information at connection time.

Sample Data Source Specifications

The exact specification of a data source is dependent upon the implementation of the ODBC driver used to access the data source and the characteristics of the data source itself. Therefore, it is important to document your requirements for the ODBC.INI file.

A data source specification for a driver for DEC Rdb might contain the following information:

```
[personnel]
driver=rdb.dll
description=Personnel database: CURLY
lastuid=smithjo
server=curly
schema=declare schema personnel filename"sys$sysdevice:
[corpdata]personnel.rdb"

[inventory]
driver=rdb.dll
description=Western Region Inventory
lastuid=smithjo
server=larry
schema-declare schema filename "sys$sysdevice:
[regionw.inventory]inventory.rdb"
```

A given driver can be referenced in more than one data source specification.

An example for MS SQL Server might contain the following:

```
[payroll]
driver=sqlsrvr.dll
lastuid=sa
```

```
database=pubs
OemAnsi=no
```

Note that for SQL Server, a data source specification in ODBC.INI maps to a server specification in the [SQLSERVER] section of WIN.INI. In this case, the data source name must be identical to the left side of the server specification entry in the WIN.INI file.

The data source specification need not contain all of the information necessary for completing a connection to a data source. Instead, the information can be used by a driver to obtain information from another source. For example, Microsoft SQL Server maintains a list of database server connections in the WIN.INI file. A WIN.INI entry for SQL Server might contain the following:

```
[sqlserver]
payroll=dbnmp3,\\server\pipe\sqlquery
```

A data source specification in the ODBC.INI file for the PUBS database accessed by an ODBC driver—via a server called PAYROLL—might contain the following:

```
[payroll]
driver=sqlsrvr.dll
lastuid=sa
database=pubs
OemAnsi=no
```

In this case, the driver uses the data source name to locate a corresponding entry in the [sqlserver] section of the WIN.INI file.

ODBC Data Source List

Whenever a data source specification is defined, the data source name must be added to the list maintained in the section called "[ODBC Data Sources]". The ODBC Data Sources section permits the list of data source names and associated specifications to be easily enumerated.

Each entry in the data sources section consists of the data source name and a short description of the DBMS product associated with the driver referenced by the corresponding data source specification.

This section in the ODBC.INI file might appear as follows, given the entries in the preceding example and a default data source specification:

```
[ODBC Data Sources]
default=SQL Server
personnel=Rdb
inventory=Rdb
payroll=SQL Server
```

How ODBC Functions Use the ODBC.INI File

Three ODBC functions access data source specifications in the ODBC.INI file.

SQLConnect

SQLConnect accepts a data source name as one of its arguments. When **SQLConnect** is called, the Driver Manager reads the data source specification that matches the data source name (DSN) argument. The Driver Manager loads the driver DLL listed in the data source specification. Each of the **SQLConnect** arguments—data source name, user ID, and authentication ID—is passed to the driver. The driver can read the data source specification in the ODBC.INI file, if necessary, to obtain additional connection information.

If the application specifies a data source name in its call to SQLConnect but there is no corresponding data source specification in the ODBC.INI file, the Driver

Manager locates the default data source specification, listed under the data source name "[default]," and loads the corresponding driver DLL. The Driver Manager passes the application-specified data source name to the driver. If there is no default data source specification, the Driver Manager returns an error.

If the application does not specify a data source name, the Driver Manager attempts to locate a default data source specification in the ODBC.INI file. If there is a default data source specification, the Driver Manager loads the driver DLL named in the default specification and passes "default" to the driver as the data source name.

If the application does not specify a data source name and no default data source specification exists, the Driver Manager returns an error.

SQLDataSources

SQLDataSources reads the [ODBC Data Sources] section of the ODBC.INI file and returns the associated list of data source names. It also reads the data source specifications that correspond to the names in the data sources section. If there is a user-defined description associated with a data source specification, **SQLDataSources** returns the description.

SQLDriverConnect

SQLDriverConnect is used as an alternative to **SQLConnect** for data sources that require connection information other than the three arguments provided by **SQLConnect**. The application specifies a data source name as part of the connection string argument of **SQLDriverConnect**.

The connection string allows an application to pass all information required by a driver to establish a connection to a specific data source. **SQLDriverConnect** can, however, prompt the user for connection information. The Driver Manager provides an optional dialog to allow the user to select a data source from a list of the data source names from the ODBC.INI file. Once the Driver Manager has a specific data source name, the Driver Manager loads the driver DLL that is listed in the corresponding data source specification.

Once the driver is loaded, the driver can display a dialog to elicit implementation-specific logon information from the user. This information depends on the requirements of the data source; it typically consists of user ID and password. The driver uses this information to replace or supplement information from the data source specification in the ODBC.INI file, or to update the data source specification. For example, after a successful connection, a driver might save the user ID for later connections to the data source.

If the application supplies a data source name but there is no corresponding data source specification in the ODBC.INI file, the Driver Manager locates the default data source specification and loads the corresponding driver DLL. The Driver Manager passes the application-supplied data source name to the driver as part of

the connection string.

If the application supplies a data source name but there is no corresponding data source specification in the ODBC.INI file and no default data source specification exists, the Driver Manager returns an error.

The ODBC SETUP Routine

The ODBC SETUP routine creates the ODBC.INI file when ODBC is first installed. The SETUP routine prompts for information to create an initial set of user-defined data source specifications. Once ODBC is installed, the user can run the SETUP routine to add, modify and delete data source specification entries from the file.

The SETUP routine uses a two-layer architecture: a top layer for generic management tasks and a lower layer for driver-specific tasks. Microsoft supplies the top layer, which supports installation, configuration, and management of drivers.

A user can run SETUP.EXE to define a default driver or select one or more drivers to install. Once the user selects a driver. the SETUP routine loads a driver-specific setup DLL. This DLL, written by the driver developer, displays a configuration dialog box that prompts the user for all relevant connection information.

Sample Application Code

XE "Application:sample code"§XE "Static SQL:example"§The following subsections contain two ODBC examples that are written in the C language:

- An example that uses static SQL functions to create a table, add data to it, and select the inserted data.
- An example of interactive ad-hoc query processing.

Static SQL Example

The following example constructs SQL statements within the application. The example includes embedded SQL calls for illustrative purposes.

```
#include "SQL.H"
#include <string.h>

#ifndef NULL
#define NULL   0
#endif

int print_err(HDBC hdbc, HSTMT hstmt);

int example1(server, uid, pwd)
UCHAR * server;
UCHAR * uid;
UCHAR * pwd;
{
HENV   henv;
HDBC   hdbc;
HSTMT  hstmt;

SDWORD  id;
UCHAR   name[51];
SDWORD  namelen;
UWORD   scale;

scale = 0;
/* EXEC SQL CONNECT TO :server USER :uid USING :authentication_string; */
SQLAllocEnv(&henv); /* allocate an environment handle */

SQLAllocConnect(henv, &hdbc);  /* allocate a connection handle */

/* connect to database */
if (SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS)
     != SQL_SUCCESS)
   return( print_err(hdbc, SQL_NULL_HSTMT) );

SQLAllocStmt(hdbc, &hstmt);  /* allocate a statement handle */

/* EXEC SQL CREATE TABLE NAMEID (ID integer, NAME varchar(50));  */
{
  UCHAR create[] ="CREATE TABLE NAMEID (ID integer, NAME varchar(50))";

/* execute the sql statement */
  if(SQLExecDirect(hstmt, create, SQL_NTS) != SQL_SUCCESS)
     return(print_err(hdbc, hstmt));
}
/* EXEC SQL COMMIT WORK; */
SQLTransact(hdbc, SQL_COMMIT);  /* commit create table */

/* EXEC SQL INSERT INTO NAMEID VALUES ( :id, :name ); */
{
    UCHAR insert[]= "INSERT INTO NAMEID VALUES (?, ?)";
```

```
/* show the use of SQLPrepare/SQLExecute method */
/* prepare the insert*/
  if(SQLPrepare(hstmt, insert, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));
  SQLSetParam(hstmt, 1,SQL_C_LONG, SQL_INTEGER, (UDWORD)sizeof(UDWORD),
      scale, (PTR)&id, (SDWORD *)NULL);
  SQLSetParam(hstmt, 2,SQL_C_CHAR, SQL_VARCHAR, (UDWORD)sizeof(name),
      scale, (PTR)name, (SDWORD *)NULL);

/* now assign parameter values and execute the insert*/
  id=500;
  (void)strcpy(name, "Babbage");
  if(SQLExecute(hstmt) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));
}
/* EXEC SQL COMMIT WORK; */
SQLTransact(hdbc, SQL_COMMIT);  /* commit inserts */

/* EXEC SQL DECLARE c1 CURSOR FOR SELECT ID, NAME FROM NAMEID; */
/* EXEC SQL OPEN c1; */
/* note that "declare c1 cursor for" is NOT specified by the app.
ExecuteDBL */
{
  UCHAR select[]= "select ID, NAME from NAMEID";
  if(SQLExecDirect(hstmt, select, SQL_NTS) != SQL_SUCCESS)
  return(print_err(hdbc, hstmt));
}
/* EXEC SQL FETCH c1 INTO :id, :name; */
/* use column binding with SQLSetParam */
SQLBindCol(hstmt, 1,SQL_C_LONG, (PTR)&id, (SDWORD)sizeof(SDWORD),
  (SDWORD *)NULL);
SQLBindCol(hstmt, 2,SQL_C_CHAR, (PTR)name, (SDWORD)sizeof(name),
  &namelen);

SQLFetch(hstmt);  /* now execute the fetch */

/* finally, we should commit, discard hstmt, disconnect */
/* EXEC SQL COMMIT WORK; */
SQLTransact(hdbc, SQL_COMMIT);    /* commit the transaction  */

/* EXEC SQL CLOSE c1; */
SQLFreeStmt(hstmt, SQL_DROP);   /* free the statement handle */

/* EXEC SQL DISCONNECT; */
SQLDisconnect(hdbc);   /* disconnect from the database */

SQLFreeConnect(hdbc); /* free the connection handle */
SQLFreeEnv(henv);   /* free the environment handle  */

return(0);
}
```

Interactive Ad-Hoc Query Example

XE "Interactive ODBC example"§XE "Results:determining (example)"§The following example illustrates how an application can determine the nature of the result set prior to retrieving results.

```
#include "SQL.h"
#include <string.h>
#include <stdlib.h>

#define  MAXCOLS   100

#define  max(a,b) (a>b?a:b)
int print_err(HDBC hdbc, HSTMT hstmt);
int build_indicator_message(UCHAR * errmsg, PTR * data, SDWORD *len);
UDWORD display_length(SWORD coltype, UDWORD collen,  UCHAR *colname);

example2(server, uid, pwd, sqlstr)
UCHAR * server;
UCHAR * uid;
UCHAR * pwd;
UCHAR * sqlstr;
{
int     i;
HENV    henv;
HDBC    hdbc;
HSTMT   hstmt;
UCHAR   errmsg[256];
UCHAR   colname[32];
SWORD   coltype;
SWORD   colnamelen;
SWORD   nullable;
UDWORD  collen[MAXCOLS];
SWORD   scale;
SDWORD  outlen[MAXCOLS];
UCHAR * data[MAXCOLS];
SWORD   nresultcols;
SDWORD  rowcount;
SWORD   rc;

SQLAllocEnv(&henv);  /* allocate an environment handle */

SQLAllocConnect(henv, &hdbc);  /* allocate a connection handle */

/* connect to database */
if (SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS,  pwd, SQL_NTS)
     != SQL_SUCCESS )
  return( print_err(hdbc,SQL_NULL_HSTMT) );

/* allocate a statement handle */
SQLAllocStmt(hdbc, &hstmt);

/* execute the SQL statement */
if(SQLExecDirect(hstmt, sqlstr, SQL_NTS) != SQL_SUCCESS)
    return( print_err(hdbc,hstmt) );

/* see what kind of statement it was */
SQLNumResultCols(hstmt, &nresultcols);
if (nresultcols == 0) {
  /* no result columns, so must be non-select */
  /* check rowcount */
  SQLRowCount(hstmt, &rowcount);
  if (rowcount > 0 ) {
     /* rowcount was affected, so must have been update, insert or delete */
    (void)printf("%ld rows affected\n", rowcount);
    }
    else {
       /* rowcount == 0, so assume it was not an update/delete and
```

```
          therefore a DDL, Grant/Revoke,  or Commit/Rollback.
          Of course  this isn't necessarily so--it could be that the
          where clause in  the update/delete did not match any rows */
        (void)printf("Operation successful\n");
        SQLTransact(hdbc, SQL_COMMIT);
    }
}
else {  /* must have result rows */
    /* display column names */
     for (i=0; i<nresultcols; i++) {
        SQLDescribeCol(hstmt, i+1, colname, sizeof(colname),
            &colnamelen, &coltype, &collen[i],  &scale, &nullable);
        /* assume there is a display_length function which computes
              correct length given the data type  */
        collen[i] = display_length(coltype, collen[i], colname);
        (void)printf("%*.*s", collen[i], collen[i], colname);
        /* allocate memory to bind column */
        data[i] = (UCHAR *) malloc(collen[i]);
        /* bind columns to program vars, converting all types to CHAR */
        SQLBindCol(hstmt, i+1,SQL_C_CHAR, data[i], collen[i], &outlen[i]);
    }
    /* display result rows */
    while((rc=SQLFetch(hstmt))!=SQL_ERROR) {
        errmsg[0] = '\0';
        if (rc == SQL_SUCCESS_WITH_INFO) {
            for (i=0; i<nresultcols; i++)
              if (outlen[i] == SQL_NULL_DATA || outlen[i] >= collen[i] )
                  build_indicator_message(errmsg, (PTR *)&data[i], collen[i], &outlen[i], i);
            (void)printf("%*.*s ", outlen[i], outlen[i], data[i]);
        } /* for all columns in this row  */
       (void)printf("\n%s", errmsg);  /* print truncation messages, if any */
    } /* while rows to fetch */
} /* else select statement */

/* free data buffers */
for (i=0; i<nresultcols; i++)  {
    (void)free(data[i]);
}

SQLFreeStmt(hstmt, SQL_DROP );  /* free statement handle */
SQLDisconnect(hdbc);  /* disconnect from database */
SQLFreeConnect(hdbc);  /* free connection handle */
SQLFreeEnv(henv);    /* free environment handle */

return(0);
}

/************************************************************
The following functions are given for completeness, but are
not relevant for understanding the database processing
nature of ODBC
************************************************************/

#define MAX_NUM_PRECISION 15
/* define max length of char string representation of number as:
      =  max(precision) + leading sign + E + exp sign + max exp length
      =  15             + 1           + 1 + 1        + 2
      =  15 + 5
*/
#define MAX_NUM_STRING_SIZE (MAX_NUM_PRECISION + 5)

UDWORD display_length(coltype, collen, colname)
SWORD   coltype;
UDWORD  collen;
UCHAR * colname;
{
switch (coltype) {

  case SQL_VARCHAR:
  case SQL_CHAR:
      return(max(collen,strlen((char *)colname)));
      break;
```

```
 case SQL_FLOAT:
 case SQL_DOUBLE:
 case SQL_NUMERIC:
 case SQL_REAL:
 case SQL_DECIMAL:
    return(max(MAX_NUM_STRING_SIZE,strlen((char *)colname)));
    break;

 case SQL_INTEGER:
    return(max(10,strlen((char *)colname)));
    break;

 case SQL_SMALLINT:
    return(max(5,strlen((char *)colname)));
    break;

 default:
    (void)printf("Unknown datatype, %d\n", coltype);
    return(0);
    break;
 }
}


int build_indicator_message(errmsg, data, collen, outlen, colnum)
UCHAR *  errmsg;
PTR *    data;
UDWORD   collen;
SDWORD * outlen;
UWORD   colnum;
{
  if (*outlen == SQL_NULL_DATA) {
       (void)strcpy((char *)data, "NULL");
       *len=4;
  }
  else {
    sprintf((char *)errmsg+strlen((char *)errmsg),
         "%d chars truncated, col %d\n", *outlen-collen+1, colnum);
    *len=255;
  }
}
```

Testing and Debugging an Application

The ODBC developer's kit provides information for implementing Windows-based ODBC applications. Windows development and debugging tools are available for the development process. Refer to the Windows development kit for additional information about testing and debugging your ODBC application.

Support

If you need technical support regarding the Microsoft ODBC Software Development Kit, you have a wide choice of support offerings, including:

- Developer Services on CompuServe The developer services area is dedicated to providing developers with access to peer support and information specific to Microsoft development products. The **Microsoft Developer Services** area (GO MSDS) offers:

- Developer Forums This set of public forums covers information on Windows SDKs, languages, tools, and utilities from a developer's perspective. For example, the Client Server forum provides information about ODBC development. Here, you can receive peer support and Microsoft Support Engineer support for general API and function questions.
- Developer Knowledge Base An up-to-date reference tool containing developer-specific technical information about Microsoft products, compiled by Microsoft Product Support Engineers.
- Software Library A collection of text and graphics files, sample code, and utilities. The entire library is keyword-searchable, and the files can be downloaded for use locally.
- Confidential Technical Service Requests Microsoft offers private (fee-based per incident) technical support to help solve more complex development problems.

For more information about signing up for a CompuServe account, call (800) 848-8199. Ask for Operator 230 to receive a $15 free usage credit for sampling the information located in the Microsoft Developer Services area.

- Microsoft Fee-Based Offerings Microsoft offers a wide range of fee-based comprehensive support plans, one which meets your specific needs. Most plans offer telephone and electronic Service Request support from knowledgable Microsoft engineers and access to the Knowledge Base/Software library, all with a Windows interface.

  For more information about ODBC support options, please call Developer Services at
  (800) 227-4679.

Appendix A SQL Grammar

The following paragraphs list the constructs that are valid in a call to **SQLPrepare**, **SQLExecute**, or **SQLExecDirect**. This grammar is not intended to restrict the SQL syntax supported by a driver. Instead, it defines a base grammar. A driver can extend this grammar to include the syntax for a specific data source.

To the left of each construct is an indicator that tells whether the construct is part of the core grammar, the minimum grammar, or both.

Elements that are part of Integrity Enhancement Facility (IEF) and are separate from the ANSI 1989 standard are presented in the following typeface and font, distinct from the rest of the grammar:

table-constraint-definition

The set of data types defined in this grammar is not necessarily supported by a specific data source; the use and syntax of each data type is database-dependent.

Element	Core	Mini-mum
alter-table-statement ::=	X	
ALTER TABLE *base-table-name* { ADD *column-identifier data-type* | ADD (*column-identifier data-type* [, *column-identifier data-type*]...) }		
create-index-statement ::=	X	
CREATE [UNIQUE] INDEX *index-name* ON *base-table-name* (*column-identifier* [ASC | DESC] [, *column-identifier* [ASC | DESC]]...)		

Element	Core	Mini-mum
create-table-statement ::= CREATE TABLE *base-table-name-1* (*column-element* [, *column-element*] ...) *column-element* ::= *column-definition* | *table-constraint-definition* *column-definition* ::= *column-identifier data-type* [DEFAULT *default-value*] [*column-constraint-definition* [, *column-constraint-definition*]...] *column-constraint-definition* ::= NOT NULL | [UNIQUE | PRIMARY KEY] (*column-identifier*[,*column-identifier*]...) | [REFERENCES *base-table-name-2 referenced-columns*] | [CHECK (*search-condition*)] *default-value* ::= *literal* | NULL | USER *table-constraint-definition* ::= UNIQUE (*column-identifier* [, *column-identifier*] ...) | PRIMARY KEY (*column-identifier* [, *column-identifier*] ...) | CHECK (*search-condition*) | FOREIGN KEY *referencing-columns* REFERENCES *base-table-name-2 referenced-columns*	X	
create-view-statement ::= CREATE VIEW *viewed-table-name* [(*column-identifier* [, *column-identifier*]...)] AS *query-specification*	X	
delete-statement-positioned ::= DELETE FROM *table-name* WHERE CURRENT OF *cursor-name*	X	
delete-statement-searched ::= DELETE FROM *table-name* [WHERE *search-condition*]	X	X

drop-index-statement ::= X

DROP INDEX *index-name*

drop-table-statement ::= X

DROP TABLE *base-table-name*

[{ CASCADE | RESTRICT }]

Element	Core	Mini-mum
drop-view-statement ::= DROP VIEW *viewed-table-name* [{ CASCADE | RESTRICT }]	X	
grant-statement ::= GRANT {ALL | *grant-privilege* [, *grant-privilege*]... } ON *table-name* TO {PUBLIC | *user-name* [, *user-name*]... } *grant-privilege* ::= DELETE | INSERT | SELECT | UPDATE [(*column-identifier* [, *column-identifier*]...)] | REFERENCES [(*column-identifier* [, *column-identifier*]...)]	X	
insert-statement ::= INSERT INTO *table-name* [(*column-identifier* [, *column-identifier*]...)] { *query-specification* | VALUES (*insert-value* [, *insert-value*]...)} *insert-value* ::= | *dynamic-parameter* | *literal* | NULL | USER	X	
revoke-statement ::= REVOKE {ALL | *revoke-privilege* [, *revoke-privilege*]... } ON *table-name* FROM {PUBLIC | *user-name* [, *user-name*]... } [{ CASCADE | RESTRICT }] *revoke-privilege* ::= DELETE | INSERT | SELECT	X	

UPDATE

REFERENCES

Element	Core	Mini-mum
select-statement ::= SELECT [ALL | DISTINCT] *select-list* FROM *table-reference* [, *table-reference*]... [WHERE *search-condition*] [GROUP BY *column-name* [, *column-name*]...] [HAVING *search-condition*] [UNION *select-statement*]... [*order-by-clause*]	X	
select-statement ::= SELECT [ALL | DISTINCT] *select-list* FROM *table-reference* [, *table-reference*]... [WHERE *search-condition*] [*order-by-clause*]		X
select-for-update-statement ::= SELECT [ALL | DISTINCT] *select-list* FROM *table-reference* [, *table-reference*]... [WHERE *search-condition*] FOR UPDATE OF *column-name* [, *column-name*]...	X	
update-statement-positioned ::= UPDATE *table-name* SET *column-identifier* = {*expression* | NULL} [, *column-identifier* = {*expression* | NULL}]... WHERE CURRENT OF *cursor-name*	X	
update-statement-searched UPDATE *table-name* SET *column-identifier* = {*expression* | NULL } [, *column-identifier* = {*expression* | NULL}]... [WHERE *search-condition*]	X	X

Elements Used in SQL Statements

The following elements are used in the SQL statements listed previously .

Element	Core	Mini-mum
approximate-numeric-literal ::=*mantissa*E*exponent* *mantissa* ::= *exact-numeric-literal* *exponent* ::= [+|-] *unsigned-integer*	X	X
approximate-numeric-type ::= FLOAT | DOUBLE PRECISION | REAL	X	X
base-table-identifier ::= *user-defined-name*	X	X
base-table-name ::= [*user-name*.]*base-table-identifier*	X	
base-table-name ::=*base-table-identifier*		X
between-predicate ::= *expression* [NOT] BETWEEN *expression* AND *expression*	X	
binary-literal ::= {implementation defined}	X	X
binary-type ::= BINARY (*length*) | VARBINARY (*length*) | LONG VARBINARY(*length*)	X	X
character ::= {any character in the implementor's character set except the newline indication}	X	X
character-string-literal :: = '{*character*}...'	X	X

character-string-type ::= CHARACTER(*length*) | CHAR(*length*) | CHARACTER VARYING(*length*) | VARCHAR (*length*) | LONG VARCHAR(*length*)	X	X
column-identifier ::= *user-defined-name*	X	X
column-name ::= [{*table-name* | *correlation-name*}.]*column-identifier*	X	
column-name ::= [*table-name*.]*column-identifier*		X
comparison-operator ::= < | > | <= | >= | = | <>	X	X
comparison-predicate ::= *expression comparison-operator* {*expression* | (*sub-query*)}	X	
comparison-predicate ::= *expression comparison-operator expression*		X

Element	Core	Mini-mum
correlation-name ::= *user-defined-name*	X	
cursor-name ::= *user-defined-name*	X	
data-type ::= *binary-type* | *character-string-type* | *date-type* | *exact-numeric-type* | *approximate-numeric-type* | *time-type* | *timestamp-type*	X	X
date-literal ::= *'date-value'*	X	X
date-separator ::= -		
date-type ::= DATE	X	X
date-value ::= *years-value date-separator months-value date-separator days-value*	X	X
days-value ::= *digit digit*	X	X
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	X	X
dynamic-parameter ::= ?	X	X
exact-numeric-literal ::= [+|-] { *unsigned-integer* [.*unsigned-integer*] | *unsigned-integer*.	X	X

| *.unsigned-integer* }

exact-numeric-type ::= X X
DECIMAL(*precision, scale*)
| INTEGER
| SMALLINT
| NUMERIC(*precision, scale*)
| TINYINT
| BIGINT
| BIT
precision ::= *unsigned-integer*
scale ::= *unsigned-integer*

exists-predicate ::= EXISTS (*sub-query*) X

Element	Core	Mini-mum
expression ::= *term* | *expression* {+|-} *term*	X	X
term ::= *factor* | *term* {*|/} *factor*	X	X
factor ::= [+|-]*primary*	X	X
primary ::= *column-name* | *dynamic-parameter* | *literal* | *set-function-reference* | USER (*2*) | (*expression*)	X	
primary ::= *column-name* | *dynamic-parameter* | *literal* | (*expression*)		X
hours-value ::= *digit digit*	X	X
index-identifier ::= *user-defined-name*	X	
index-name ::= [*user-name*.]*index-identifier*	X	
in-predicate ::= *expression* [NOT] IN {(*value* {, *value*}...) | (*sub-query*)} *value* ::= *literal* | USER | *dynamic-parameter*	X	
join-condition ::= ON *search-condition*	X	X
keyword ::= (see list of reserved keywords)	X	X

length ::= *unsigned-integer*	X	X
letter ::= *lower-case-letter* | *upper-case-letter*	X	X
like-predicate ::= *column-name* [NOT] LIKE *pattern-value*	X	X
pattern-value ::= *character-string-literal* | *dynamic-parameter* | USER	X	
pattern-value ::= *character-string-literal* | *dynamic-parameter* (in character-string-literal, the percent character ('%') matches 0 or more of any character; the underscore character ('_') matches 1 or 0 characters)		X
literal ::= *character-string-literal* | *numeric-literal*	X	X
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	X	X
minutes-value ::= *digit digit*	X	X
months-value ::= *digit digit*	X	X
null-predicate ::= *column-name* IS [NOT] NULL	X	X

Element	Core	Mini-mum
numeric-literal ::= *exact-numeric-literal* | *approximate-numeric-literal*	X	X
order-by-clause ::= ORDER BY *sort-specification* [, *sort-specification*]... *sort-specification* ::= {*unsigned-integer* | *column-name* } [ASC | DESC]	X	X
outer-join ::= *table-reference* LEFT OUTER JOIN *table-reference join-condition* *join\-condition* ::= ON *search-condition* (Notes: For outer joins, *search-condition* must contain only the join condition between the specified *table-reference*s. The outer-join syntax must be placed within an escape clause.)		
predicate ::= *between-predicate* | *comparison-predicate* | *exists-predicate* | *in-predicate* | *like-predicate* | *null-predicate* | *quantified-predicate*	X	
predicate ::= *comparison-predicate* | *like-predicate* | *null-predicate*		X
quantified-predicate ::= *expression comparison-operator* {ALL | ANY} (*sub-query*)	X	
referenced-columns ::= (*column-identifier* [, *column-identifier*]...)	X	
referencing-columns ::= (*column-identifier* [, *column-identifier*]...)	X	
search-condition ::= *boolean-term* [OR *search-condition*] *boolean-term* ::= *boolean-factor* [AND *boolean-term*] *boolean-factor* ::= [NOT] *boolean-primary*	X	X

boolean-primary ::= *predicate* | (*search-condition*)

seconds-fraction ::= *digit digit digit [digit digit digit]*	X	X
seconds-value ::= *digit digit*	X	X
select-list ::= *	*select-sublist* [, *select-sublist*]...	X	X
select-sublist ::= *expression*	{*table-name*	*correlation-name*}.*	X	
select-sublist ::= *expression*		X
separator ::=	X	X

The blank character or an implementation-defined end-of-line indicator.

Element	Core	Mini-mum
set-function-reference ::= COUNT(*) | *distinct-function* | *all-function* *distinct-function* ::= {AVG | COUNT | MAX | MIN | SUM} (DISTINCT *column-name*) *all-function* ::= {AVG | MAX | MIN | SUM} (*expression*)	X	
SQL-*escape-clause* ::= *standard*-SQL-*escape-initiator extended*-SQL-*text standard*-SQL- *escape-terminator* | *extended*-SQL-*escape-prefix extended*-SQL-*text extended*-SQL- *escape-terminator* *standard*-SQL-*escape-initiator* ::= *standard*-SQL-*escape-prefix* SQL- *escape-identification*, *standard*-SQL-*escape-prefix* ::= --*(*extended*-SQL-*escape-prefix* ::= { *standard*-SQL-*escape-terminator* ::= --*) *extended*-SQL-*escape-terminator* ::= } SQL-*escape-identification* ::= SQL-*escape-vendor-clause* SQL-*escape-vendor-clause* ::= VENDOR(M*icrosoft*), PRODUCT(ODBC)	X	X
sub-query ::= SELECT [ALL | DISTINCT] *select-list* FROM *table-reference* [, *table-reference*]... [WHERE *search-condition*] [GROUP BY *column-name* [, *column-name*]...] [HAVING *search-condition*]	X	
table-identifier ::= *user-defined-name*	X	X
table-name ::= [*user-name*.]*table-identifier*	X	

table-name ::=*table-identifier*		X
table-reference ::= *table-name* [*correlation-name*]	X	
table-reference ::= *table-name*		X
time-literal ::='*time-value*'	X	X

Element	Core	Mini-mum
time-separator ::= :		
time-type ::= TIME	X	X
time-value ::= *hours-value time-separator minutes-value time-separator seconds-value*	X	X
timestamp-literal ::='*date-value*:*time-value*.[*seconds-fraction*]'	X	X
timestamp-type ::= TIMESTAMP	X	X
token ::= *delimiter-token* | *non-delimiter-token* *delimiter-token* ::= *character-string-literal* | , | (|) | < | > | . | : | = | * | + | - | / | <> | >= | <= | ? *non-delimiter-token* ::= *keyword* | *numeric-literal* | *user-defined-name*	X	X
unsigned-integer ::= {*digit*}...	X	X
upper-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	X	X
user-defined-name ::= *letter*[*digit* | *letter* | _]...	X	X
user-name ::= *user-defined-name*	X	X

viewed-table-identifier ::= *user-defined-name*	X	
viewed-table-name ::= [*user-name*.]*viewed-table-identifier*	X	
years-value ::= *digit digit digit digit*	X	X

List of 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.

ABSOLUTE
ADA
ADD
ALL
ALLOCATE
ALTER
AND
ANY
ARE
AS
ASC
ASSERTION
AT
AUTHORIZATION
AVG
BEGIN
BETWEEN
BIT
BIT_LENGTH
BY
CASCADE
CASCADED
CASE
CAST
CATALOG
CHAR
CHAR_LENGTH
CHARACTER
CHARACTER_LEN
GTH
CHECK
CLOSE
COALESCE
COBOL
COLLATE
COLLATION
COLUMN
COMMIT
CONNECT
CONNECTION
CONSTRAINT
CONSTRAINTS
CONTINUE
CONVERT
CORRESPONDING
COUNT
CREATE
CURRENT
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMEST
AMP
CURSOR
DATE
DAY
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFERRABLE
DEFERRED
DELETE
DESC
DESCRIBE
DESCRIPTOR
DIAGNOSTICS
DICTIONARY
DISCONNECT
DISPLACEMENT
DISTINCT
DOMAIN
DOUBLE
DROP
ELSE
END
END-EXEC
ESCAPE
EXCEPT
EXCEPTION
EXEC
EXECUTE
EXISTS
EXTERNAL
EXTRACT
FALSE
FETCH
FIRST
FLOAT
FOR
FOREIGN
FORTRAN
FOUND
FROM
FULL
GET
GLOBAL
GO
GOTO
GRANT
GROUP
HAVING
HOUR
IDENTITY
IGNORE
IMMEDIATE
IN
INCLUDE
INDEX
INDICATOR
INITIALLY
INNER
INPUT
INSENSITIVE
INSERT
INTEGER
INTERSECT
INTERVAL
INTO
IS
ISOLATION
JOIN
KEY
LANGUAGE
LAST
LEFT
LEVEL
LIKE
LOCAL
LOWER
MATCH
MAX
MIN
MINUTE
MODULE
MONTH
MUMPS
NAMES
NATIONAL

NCHAR
NEXT
NONE
NOT
NULL
NULLIF
NUMERIC
OCTET_LENGTH
OF
OFF
ON
ONLY
OPEN
OPTION
OR
ORDER
OUTER
OUTPUT
OVERLAPS
PARTIAL
PASCAL
PLI
POSITION
PRECISION
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVILEGES
PROCEDURE
PUBLIC
RESTRICT
REVOKE
RIGHT
ROLLBACK
ROWS
SCHEMA
SCROLL
SECOND
SECTION
SELECT
SEQUENCE
SET
SIZE
SMALLINT
SOME
SQL
SQLCA
SQLCODE
SQLERROR
SQLSTATE
SQLWARNING
SUBSTRING
SUM
SYSTEM
TABLE
TEMPORARY
THEN
TIME
TIMESTAMP
TIMEZONE_HOUR
TIMEZONE_MINUT
E
TO
TRANSACTION
TRANSLATE
TRANSLATION
TRUE
UNION
UNIQUE
UNKNOWN
UPDATE
UPPER
USAGE
USER
USING
VALUE
VALUES
VARCHAR
VARYING
VIEW
WHEN
WHENEVER
WHERE
WITH
WORK
YEAR

Index