Driver Developer's Guide

§

Contents

§

§

Introduction

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

- What is the ODBC interface?
- What features does the interface offer?
- How does a driver developer implement ODBC functions?

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 use ODBC functions.
- Chapter 9, “Constructing an ODBC Driver,” lists developer's kit contents and describes how to build a driver.

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 the use of the ODBC interface from an application, refer to the *ODBC Application Programmer's Guide*.

Audience

The ODBC software development kit (SDK) 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 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 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 back-end database management systems. An application developer can develop, compile, and ship an application without targeting a specific DBMS product. Users can then add modules called database *drivers* that link the 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, an application includes the statement as an argument in an ODBC function call. The application need not customize the statement for a specific DBMS.

  Multiple-tier ODBC drivers should not constrain applications from sending DBMS-specific SQL syntax, but drivers must translate SQL syntax when it conflicts with the syntax or semantics defined in Appendix A, "SQL Grammar." For example, Microsoft SQL Server version 1.1 uses != (instead of <>) to represent "does not equal." A driver written for Microsoft SQL Server must translate <> to !=.

  To ensure maximum interoperability, ODBC *applications* are encouraged to use only the syntax defined in Appendix A, "SQL Grammar."
- 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:

- An application can construct SQL statements at compile time or at run time.
- An application can use the same object code to access different DBMS products.
- An application can connect to multiple instances of DBMS products.
- An application can ignore underlying data communications protocols associated with the DBMS product.
- An application can send and receive data in a format convenient to the

§

application; the driver converts data values to the format of a specific DBMS product.

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.

Each driver implements a set of ODBC functions.

The following sections describe the ODBC architecture in more detail.

ODBC Components

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 a DBMS, the operating system the DBMS runs on, and the network (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 tasks:

- Uses the ODBC.INI file to map a data source name (provided by the application) 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 the application requests access to a cursor name.)
- Initiates transactions if the data source requires explicit transaction initiation. (This operation is invisible to the application.)

Data Source

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 combination of DBMS product, remote operating system, and network as a data source.

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

- 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.
- 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.

A single-tier driver can restrict the set of SQL statements that the application can submit. 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–one standalone and one that uses a network.

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.

Configuration Examples

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 might reside in a network.

Applications can also communicate across wide area networks:

Implementing Sets of Functionality

XE "ODBC:functionality sets"§One of the strengths of the ODBC interface is interoperability: a developer can create an ODBC application without targeting a specific data source. Users can add drivers to the application after the application is compiled and shipped.

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, data sources provide a varying range of functionality. Therefore, the ODBC interface defines conformance designations, which determine the set of ODBC functions and SQL statements supported by a driver.

ODBC Function Call Support

Each ODBC driver supports a set of core ODBC functions and data types 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 set of core data types supported by a driver depends upon the set of data types supported by the data source.
- Extended functions and data types support additional features, including date, time, and timestamp data type literals, scrollable cursors, and asynchronous execution of function calls. Extended functions may not be supported by a specific driver. Extended functions are divided into two conformance designations, Level 1 and Level 2, each of which is a superset of the core functions.

§

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 CAE draft

specification (1991).

In addition to the core and minimum sets, ODBC defines SQL syntax for date 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 support a set of functionality depends on the set of features the data source supports. The use of some extended functions, if available, can enhance performance.

Driver writers are encouraged to implement as many ODBC functions as possible, to ensure the widest possible use by applications.

Chapters 3 through 8 describe how to implement ODBC functions. The *ODBC API Reference*, Chapter 2, lists all ODBC functions in alphabetic order.

Connections and Transactions

Before an application can communicate with a data source, it requests a connection. If the connection is successful, the driver returns a value to the application for subsequent use when calling ODBC functions. This value is known as a connection handle.

An application can 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"§A driver maintains a transaction for each active connection. The application can request that each SQL statement be automatically committed upon completion; otherwise, the driver waits for an explicit commit or rollback request from the application. When the driver performs a commit or rollback operation, the driver resets all statement requests associated with the connection.

An application can switch connections while transactions are in progress on the current connection. The driver manages this work.

If the driver does not implement extended functionality, an application can switch connections only after it requests a commit or roll back operation for the transaction on the current 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 the database environment, 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 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 Implementing ODBC Functions

Each driver implements a set of ODBC functions. This work includes allocating and deallocating memory, transmitting or processing SQL statements, and returning results and errors.

This chapter describes characteristics of ODBC functions and includes the following information:

- Elements of ODBC functions.
- Processing SQL statements.
- Returning results.

Elements of ODBC Functions

The following characteristics apply to all ODBC functions.

General Information

XE "ODBC:functions, general information"§Each ODBC function name starts with the prefix SQL. Each function includes one or more arguments. Arguments are defined for input (to the driver) or output (from the driver). You can include variable-length data where appropriate.

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

The ODBC interface defines valid data types, including C data types and SQL data types. 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.

ODBC accepts the following lengths XE "SQL_NULL_DATA"§XE "SQL_NTS"§for input arguments:

- 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.

The application allocates memory for output buffers. Therefore, the application must indicate the length of each output buffer. On output, the driver tells the application how much data was actually stored in the buffer. Each output argument has two associated 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.

If the application uses null terminated strings and passes a null pointer for the output length, the driver does not return the length.

For more information about error names and other #defined constants, refer to the header file listed in the *ODBC API Reference*, Appendix F, “Sample 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 a 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. One or more of these handles are required by all 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 connection handles, refer to Chapter 5, “Establishing Connections.” For more information about statement handles, refer to Chapter 6, “Processing an SQL Statement.”

Processing SQL Statements

An application submits an SQL statement as an argument in an ODBC function call. 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:

- C data types describe how data is stored in your C program.
- 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 support only the core set of ODBC functions, map all data to core data types.
  - Extended data types support data types from many current DBMS products.

Each ODBC data type has a corresponding C data type. These data types are

defined in the ODBC header files, listed in the *ODBC API Reference*, Appendix F, "Sample C 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.”

XE "SQLDescribeCol:providing data type information"§ XE "SQLGetTypeInfo:providing data type information"§A driver supports ODBC data types in the following ways:

- Accepts ODBC data types as arguments in function calls.
- Translates ODBC data types to types acceptable by the data source, if necessary.
- Returns results as valid data type arguments.
- Provides access to data type information through the **SQLDescribeCol** function and, optionally, the **SQLGetTypeInfo** function.

Returning 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 an application calls an ODBC function, the driver returns a predefined status code that indicates success or failure. These status codes indicate success, warning, or failure status. The application then calls the **SQLError** function, if necessary, to retrieve detailed information about a warning or failure. 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. The application calls **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 environment, handle, connection handle, or statement handle. This indicates a programming error. No

	further information is available from **SQLError**.
SQL_STILL_EXECUTING	A function that was started asynchronously is still executing.
SQL_NEED_DATA	While processing a 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, the application calls **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 or message information.

SQLError itself never returns an error or message value.

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

XE "Processing results"§XE "Results:processing"§An application obtains information about results through the following mechanisms:

- The return code.
- A call to **SQLRowCount**.

- A call to **SQLNumResultCols**.
- A call to **SQLDescribeCol**.
- A call to **SQLColAttributes**.
- A call to **SQLNumParams**.

If an operation does not affect or return rows, such as an SQL **GRANT** or **REVOKE** operation, the application checks the return code to determine the outcome of the operation. If the operation affected rows, the application retrieves the row count to determine the outcome of the operation. If the request was a **SELECT** query, the application checks the number of result columns and data descriptions to gain information about the result set.

For more information about handling results, refer to Chapter 7, “Returning Results.”

4 Application Use of the ODBC Interface

This chapter describes the flow of calls that an application sends to a driver. This information is included for reference purposes. The following chapters describe how to use the ODBC interface to write a driver.

To communicate with the driver and gain access to a data source, an application follows a general sequence of steps:

1. Establishes a connection to the driver, specifying a data source name and additional connection information.
2. For each SQL request:
   - Places the SQL text string into a buffer.
   - Submits the SQL string for execution.
   - Inquires about the results. If the operation was unsuccessful, processes error information.
   - Fetches data row by row, if available.
3. When finished with a transaction, the application requests a commit or rollback operation for **INSERT**, **UPDATE**, or **DELETE** operations.
4. When finished submitting statements to the data source, the application terminates 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.

Additional Information

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

The *ODBC API Reference* lists syntax and usage information for each ODBC function.
B of the reference manual lists valid command flow sequences.

5 Establishing Connections

XE "Connections:establishing"§This chapter describes how an application, Driver Manager, and driver establish a connection to a target data source.

Using the Driver Manager

XE "Driver Manager:communicating with"§ODBC function calls come through the Driver Manager to your driver. When an application calls 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.

Establishing a Connection to a Data source

All drivers must support the following connection-related functions:

- **SQLAllocEnv** allows the driver to allocate storage for environment information.
- **SQLAllocConnect** allows the driver to allocate storage for connection information.
- **SQLConnect** allows an application to establish a connection with the data source. The application passes the following information in the 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 Driver."
  - 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 remote data source (optional).

The Driver Manager does not load a driver or call the driver's **SQLAllocEnv** or **SQLAllocConnect** functions until the application calls **SQLConnect** or **SQLDriverConnect** and specifies a data source name. Until that point, the Driver Manager works with its own handles and manages connection information. Once the application calls **SQLConnect** or **SQLDriverConnect**, the Driver Manager calls the driver's **SQLAllocEnv** and **SQLAllocConnect** routines and calls either **SQLConnect** or **SQLDriverConnect**, respectively. The driver then allocates handles and initializes itself.

The following diagram shows the underlying command flow between the Driver Manager and the driver.

Before the Driver Manager can access a driver, it must have access to information in the ODBC.INI file. For additional information about this file, refer to Chapter 9, "Constructing an ODBC Driver."

Supporting ODBC Functions

ODBC 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. The application can call **SQLGetFunctions** to determine if the driver supports a particular function.

The ODBC API Reference, Chapter 1, "ODBC Function Summary," lists conformance designations 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.

SQLGetInfo allows an application to determine the conformance designation supported by a driver.

Extensions for Processing Connections

The following table lists connection-related extended functions. The paragraphs following this 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 "SQLSetOption"§XE "SQLGetInfo"§XE "SQLGetTypeInfo"§

Function Name	Description
SQLDataSources	Returns a list of available data sources. The Driver Manager retrieves this information from the ODBC.INI file.
SQLDriverConnect	Asks the Driver Manager or driver to obtain login information from a user. The Driver Manager uses this information to establish a connection with a driver.
SQLGetFunctions	Returns information about functions supported by a driver. This function allows an application to determine whether a driver supports a function.
SQLGetInfo	Returns general information about a driver and data source, including file names, versions, and the maximum length of names supported by the data source.
SQLGetTypeInfo	Returns the data types supported by a driver and data source.
SQLSetConnectOption **SQLGetConnectOption** **SQLSetStmtOption** **SQLGetStmtOption**	Set and return operational parameters for the driver and data source. Options include access mode, timeout values, implicit commit operation, and asynchronous execution of ODBC functions.

6 Processing an SQL Statement

An 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 SQL statements that can be prepared in embedded SQL.

s6

§

Note

The application cannot submit **COMMIT** and **ROLLBACK** SQL statements. Instead, an application calls **SQLTransact** to request a commit or rollback operation.

§

s11

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

Note that any SQL statement can be executed with either the **SQLPrepare** and **SQLExecute** sequence or the **SQLExecDirect** command, depending on whether the application plans to submit the SQL statement once or more than once. This functionality differs from embedded SQL, since 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
s6

s11

This chapter describes ODBC functions that support SQL statement processing. For information about results, refer to Chapter 7, “Returning Results.”
s11

Allocating a Statement Handle
XE "Statement handles:establishing"§XE "Handles:establishing statement"§Before an application can submit an SQL statement, it must request a statement handle. The application requests a statement handle by calling **SQLAllocStmt**. The application includes a connection handle as an argument in the **SQLAllocStmt** call. The driver allocates storage for the statement, associates the statement with the referenced connection, and returns the handle to the application.
A driver uses the statement handle to reference storage for names, parameter and bind information, error messages, and other information related to a statement processing stream.
s11
Assigning Storage for Results (Binding)
XE "Binding result columns"§XE "Results:binding"§XE "Results:allocating storage for"§An application can assign storage for result columns before or after submitting an SQL statement.
XE "SQLBindCol"§ A driver supports result storage through the **SQLBindCol** function, as follows:
- Accepts pointer arguments that reference storage areas.
- Checks to make sure the pointers are not null.
- Associates each column with the given storage area.
- Stores information about whether the application wants the driver to convert the results to a different data type.

The driver uses this information during subsequent fetch operations.
s11
Providing Prepared or Direct Execution
XE "Prepared execution"§XE "Direct execution"§XE "SQL statements:executing"§An application has two execution options for SQL requests:
- Prepared Execute the same statement more than once without respecifying the SQL string. This option, if supported by the data source, also allows the application to obtain information about the result set prior to execution.
- Direct Submit the statement once without obtaining the result format prior to execution.

s6
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."
s6

Prepared Execution
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"§To support prepared execution, implement the following three function calls. These calls can appear in any order after **SQLAllocStmt** and prior to **SQLExecute**:

- **SQLPrepare** Submit the requested SQL statement to the data source.
- **SQLSetParam** If the application includes parameter markers in the SQL statement, the application must call **SQLSetParam** to assign a storage area to each corresponding parameter marker. The application need not call **SQLSetParam** for each execution of the same SQL statement; if this is not the first execution of an SQL statement, an application can reuse previous storage areas.
- **SQLSetCursorName** Associate a cursor name with your prepared request. **SQLSetCursorName** is useful if the application plans to issue a positioned update or delete (**UPDATE WHERE CURRENT OF** *cursor-name* or **DELETE WHERE CURRENT OF** *cursor-name*).

After setting the cursor name with **SQLSetCursorName** or implicitly (by executing a **SELECT** statement), the application can call **SQLGetCursorName** to retrieve the cursor name for subsequent positioned UPDATE and DELETE statements.
If the application does not call **SQLSetCursorName**, the driver is responsible for generating a cursor and performing operations with the cursor.
s6
Once the application establishes all necessary names and parameters, the application calls **SQLExecute**. After retrieving the results, the application can assign new parameter values and resubmit the SQL statement.
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 you 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 return result descriptors (information about the format of the result set) prior to executing the SQL statement.

Prepared execution may not be available for single-tier implementations. In this case, the driver may need to generate the equivalent of a **SELECT** statement with no result rows to obtain information about result columns.
An application cannot submit a **COMMIT** or **ROLLBACK** statement in a call to **SQLExecute**. Instead, the application must call **SQLTransact** to request a **COMMIT** or **ROLLBACK** operation.
s6
Direct Execution

XE "SQLExecDirect"§The **SQLExecDirect** function supports direct execution of SQL statements. The driver sends the SQL string directly to the data source. The application should use **SQLExecDirect** if it does not require result descriptors prior to execution and if the statement is executed only once.
s11

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 an application submits a **SELECT** statement that returns multiple rows and the driver returns one or more result rows, the application can request a positioned **UPDATE** or **DELETE** to update or delete the row referenced by the cursor.

An application can use one of the following options to perform a positioned **UPDATE** or **DELETE**:

- Prepared execution The application calls **SQLPrepare** with szSqlStr set to the text of the positioned **UPDATE** or **DELETE** statement. The application uses a different hstmt than that used for the **SELECT** statement, and includes the cursor name associated with the earlier **SELECT** statement. The application then sets parameter values as necessary and calls **SQLExecute** to submit the statement.
- Direct execution The application calls **SQLExecDirect** with szSqlStr set to the text of the **UPDATE** or **DELETE** statement. The application uses a different hstmt than that used for the **SELECT** statement. The application includes the cursor name associated with the earlier **SELECT** statement.

s6

To support this functionality, a driver must associate the new hstmt with the existing hstmt, submit the positioned **UPDATE** or **DELETE** statement, include any parameter values associated with the statement, and return 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; an application could include calls to **SQLSetParam** in either sequence, or combine prepared and direct requests in the same processing stream.

s6

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; processing arrays of parameters; sending large data values; implementing scrollable cursors; asynchronous processing; and processing scalar functions, extended data types, and outer joins.
s/6
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.

SQLParamData, **SQLPutData**
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.

s11
Obtaining Information about the Data
To return information about data, implement the following functions:

- **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 comprise 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. The application fetches these results in the same manner as it retrieves query results (through a call to **SQLFetch**).
If the data source associated with a driver does not support data dictionary functions, the driver can, optionally, implement these functions.
s11
Sending Large Data Values
XE "Large data values:sending"§To support the ability to send large data values, implement the following three functions:

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

An application first indicates that it plans to send a large data value when it calls **SQLSetParam**. In this call, the application sets pcbValue to SQL_LONG_DATA for the parameter that will receive the large value.
In the call to **SQLSetParam**, the application sets rgbValue to a value that, at run time, references the location of the data. The driver must store this value and return it 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**). It also processes intervening parameters.

- **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*.

s11

Accepting Arrays of Parameter Values

XE "SQLParamOptions"§To support specification of multiple sets of parameter values for a single SQL statement, implement the **SQLParamOptions** function. For data sources that support multiple parameter values for a single SQL statement, **SQLParamOptions** can provide performance benefits. The application can then set up an array of values and submit, for example, a single INSERT statement.

s11

Supporting 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 name is Jones. This set of information, called a result set, can contain zero, one, or more than one row.

Applications retrieve single rows by calling **SQLFetch** to move the cursor to the next row in the result set and retrieve the row. The cursor, managed by the driver, points to the current row in the result set.

s6

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. A driver can provide 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.

s6

An application calls **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, the application calls **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, the application can 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.
Unless the application calls **SQLSetCursorName** and specifies a name, the driver must generate a cursor name for fetch operations . The application can call **SQLGetCursorName** to retrieve an application-generated or driver-generated cursor name.
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.
s6
Requesting Asynchronous Processing
XE "SQL statements:processing asynchronously"§XE "Asynchronous processing"§By default, a driver processes ODBC functions synchronously; the driver does not return control to the application until a function call completes. If your driver supports asynchronous processing, however, the application 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

s6
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, an application calls **SQLSetStmtOption** and specifies ON or OFF for the SQL_ASYNC_ENABLE option. To check the setting of the SQL_ASYNC_ENABLE option, the application calls **SQLGetStmtOption**.
The application can call **SQLSetConnectOption** to enable or disable asynchronous processing for all hstmts associated with an hdbc.
Upon successful initiation of an asynchronously-initiated function, the driver returns SQL_STILL_EXECUTING. If the application resubmits the function call, return SQL_STILL_EXECUTING until the function completes. Once the function completes, return a standard return code.
s11
Processing 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.

s6
s6
§
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.
§
s6
s11
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.

s6
An application 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 or 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."

s6

An application can specify an escape clause with a canonical function in place of a DBMS-specific scalar function in an SQL statement. For detailed syntax information, 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. The application can 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."

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* **--*)**

s11

Join-type specifies a type of outer join. For example, to perform an outer join between two tables named EMPLOYEE and DEPT, using the DEPTID 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
```

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

This syntax is a subset of ANSI SQL2 outer join syntax. If you choose to support outer joins, your driver must support the specified escape sequence syntax and must scan SQL strings for embedded escape text.

7 Returning Results

An application may or may not know the form of an SQL statement prior to execution. Therefore, drivers support functions that allow an application to request information about the result set.

For operations that return more than a return code, an application calls ODBC functions to obtain the results.

The following paragraphs describe these steps.

s11

Determining Characteristics of a Result Set

XE "SQLNumResultCols"§XE "SQLDescribeCol"§XE "SQLRowCount"§Each driver supports the following three core functions:

- **SQLNumResultCols** returns the number of columns in the result set. If you return a nonzero value, the operation was a **SELECT** statement.
- **SQLDescribeCol** provides information about a column in the result set.
- **SQLRowCount** returns the number of rows influenced by the SQL statement. If you return a nonzero value, the operation was an **INSERT**, **UPDATE**, or **DELETE** operation.

s11

Returning Result Data

A driver obtains result information from the data source. An application requests this information in several ways.

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

If the application requested an **INSERT**, **UPDATE**, or **DELETE** operation, the application calls **SQLRowCount** to determine the number of rows affected by the operation. Return the row count as an argument in **SQLRowCount**.

XE "SQL statements:processing results"§XE "SELECT (SQL statement):storing results"§XE "SQLFetch:retrieving result rows"§If the application requested a **SELECT** statement, the application may call **SQLNumResultCols** and **SQLDescribeCol** to obtain information about result columns. If the application did not bind the columns to storage locations prior to execution, the application must call **SQLBindCol** prior to retrieving results. The application then calls **SQLFetch** to retrieve each row of results. The application can rebind the columns after calling **SQLFetch**.

The following diagram shows the flow of the preceding two operations:
s6

s11
s11

Returning Error and Status Information
XE "SQLError"§XE "Errors, returning"§If you detect an error while processing an ODBC call, return an appropriate return code. For example, to convey additional information even though you processed the request successfully, return SQL_SUCCESS_WITH_INFO. The application can then call **SQLError** to find out more information.
If a function fails and there is no specific error message for the situation, the driver should return the ODBC general error (80 000).
SQLError returns standardized messages and data source-specific messages. The application is responsible for reading these messages and taking appropriate action.
Each function description in the *ODBC API Reference* includes a list of valid error codes for the function. The text associated with each error is listed for references purposes, but does not prescribe exact text for the error.
For a list of 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 contain additional information about **SQLGetData**, **SQLExtendedFetch**, and returning multiple row sets.
s/6
Function Name
Description

§

SQLNativeSql
Return the SQL statement syntax for the specific data source, with escape sequences translated to native SQL code.

SQLDescribeParam
Return 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 data stored in large data types.

SQLExtendedFetch
Fetch a block of result data.

SQLSetPos
Position a cursor within a fetched block of data.

SQLScrollOptions
Request a scrollable cursor.

s11
Retrieving Data in Large Columns
XE "SQLGetData"§**SQLGetData** retrieves one column of data into a buffer. If an application calls **SQLGetData** to retrieve the column, it must not call **SQLBindCol** for the column. In addition, **SQLGetData** retrieves data only from columns to the right of the rightmost bound column.
If you choose to support **SQLGetData**, you must implement it to cooperate with **SQLBindCol** and **SQLFetch**:
n An application can call **SQLGetData** to retrieve each column in a row of a result set. The application must call **SQLFetch** to move from row to row.

- An application can combine **SQLGetData** access and **SQLFetch** access. To do so, the application must bind the columns that will be retrieved using **SQLFetch**. These columns must be contiguous. The application calls **SQLFetch** to move to a row and fetch bound columns, and then calls **SQLGetData** to retrieve each remaining column.

s/6

The application can call **SQLGetData** multiple times for the same column, if necessary.

Returning Multiple Result Sets

XE "Stored procedures, returning results"§XE "Batched SQL statements, results"§XE "SQL statements:batched processing"§Three types of SQL statements can return multiple result sets:

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

s6

To indicate that there are no more results, return the SQL_NO_DATA_FOUND return code.

The **SQLMoreResults** function processes multiple result sets.

s11

Returning Blocks of Results

If you support scrollable cursors, an application can retrieve blocks of result rows. For more information about scrollable cursors and processing blocks of result data, refer to "Supporting 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 allows applications to request termination of SQL transactions, statement-processing connections (hstmts), connections (hdbcs), and environment connections (henvs).

Terminating Statement Processing

XE "SQLFreeStmt"§The **SQLFreeStmt** function releases resources associated with a statement handle. The **SQLFreeStmt** function has four options:

- **SQL_CLOSE** Close the cursor, if one exists, and discard pending results. The application 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.

s11

Terminating Transactions

XE "SQLTransact"§The **SQLTransact** function requests a commit or rollback operation for the current transaction. The driver must submit a commit or rollback request for all operations associated with the specified hdbc; this includes operations for all hstmts associated with the hdbc.

s11

Terminating Connections

XE "Connections:terminating"§XE "SQLFreeConnect:releasing a connection handle"§XE "SQLDisconnect:closing a connection"§To allow an application to terminate the connection to your driver and the data source, provide the following three functions:

1. **SQLDisconnect** Closes a connection. The application can then use the handle to reconnect to the same data source or to a different data source.
2. **SQLFreeConnect** Releases the connection handle and frees all resources associated with the handle.
3. **SQLFreeEnv** Releases the environment handle and frees all resources associated with the handle.

9 Constructing an ODBC Driver

XE "Developer's Kit, contents"§This chapter describes how to construct a driver, starting with a description of the contents of the developer's kit and including information about implementation, installation, testing, and debugging.

s11

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

s11

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 autotests.

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
(continued)
ECONNECT.C
Sample driver implementation of: **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\DEBUG

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 the 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.
s11
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:

- A data source specification defines a data source accessible through ODBC. 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 data source list section contains a list of all valid data source names (specified in data source specifications).

s11

Data Source Specification

XE "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 [SQL SERVER] 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:
[sql server]
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 [sql server] 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 enumerated easily.
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.INI file and returns the data source name (section name) of each data source specification. If there is a user-defined description associated with a data source specification, **SQLDataSources** also 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.

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 Driver Code
XE "Driver:sample code"§The ODBC developer's kit contains a sample driver that is written in the C programming language. The sample driver is a basic driver that returns meaningful data from a subset of core and extended ODBC calls.
The sample driver uses a small, in-memory database, which contains a single table called TABLE. TABLE has two columns:

- COL_1, type SQL_VARCHAR(32)
- COL_2, type SQL_INTEGER

TABLE has three rows, as follows:
Row Number
COL_1
COL_2

§

1
Data #1
1

2
Data #2
2

3
Data #3
3

s6
If you are testing only the core ODBC calls, this database is the only visible database. Several of the extended functions create in-memory databases for specific purposes.
The following table describes how core functions are implemented in the sample driver.
Function
Description

§

SQLAllocEnv
Allocates an environment handle (henv).

SQLAllocConnect
Allocates an hdbc that will be passed to subsequent calls.

SQLConnect
Makes sure that there is no active connection to this hdbc . If there is an active connection, it returns an error.

SQLDisconnect
If there is no active connection, this routine returns an error.

SQLFreeConnect
Deallocates an hdbc that was created using **SQLAllocConnect**. If there is an active connection, **SQLFreeConnect** returns an error.

SQLAllocStmt
Allocates an hstmt that will be passed to subsequent calls.

SQLPrepare
Dummy procedure.

SQLSetParam
Dummy procedure.

SQLSetCursorName
Remembers a cursor name.

SQLGetCursorName
Retrieves the cursor name that was supplied in **SQLSetCursorName**.

SQLExecute
Performs a dummy query (SELECT * FROM TABLE). **SQLExecute** does not care what the query is; it always returns the same data.

SQLExecDirect
Passes any parameters along to **SQLPrepare**, and then passes them to **SQLExecute** if **SQLPrepare** doesn't fail.

Function
Description

§

SQLError
This is one of the more complicated core functions. Whenever a call fails, the failing function adds a small error block to a chain that is attached to either an hstmt or an hdbc. Successive calls to **SQLError** return information about these error blocks.

SQLCancel
Calls **SQLFreeStmt** with the SQL_CLOSE option.

SQLNumResultCols
Returns the number of columns in the in-memory database that are associated with an hstmt .

SQLDescribeCol
Returns information about columns in the in-memory database associated with an hstmt .

SQLBindCol
Stores the column number and address. The next time the application calls **SQLFetch**, the driver stores data in the specified location.

SQLFetch
Advances the cursor and stores data into any addresses specified in a call to **SQLBindCol**.

SQLRowCount
Returns the number of rows in the in-memory database. Since the database is in memory, the driver can determine this number.

SQLFreeStmt
Deallocates of an hstmt and frees associated memory.

SQLFreeEnv
Frees a previously-allocated henv.

SQLTransact
Dummy function.

The extended functions are implemented as follows:
Function
Description

§

SQLDriverConnect
Displays a dialog box with an empty listbox in it. If the user selects OK, **SQLDriverConnect** calls **SQLConnect** and returns this string:
DSN=SAMPLE;SRV=SAMPLE;UID=SA;PWD=NOTHING

SQLGetInfo
Attaches an in-memory database to an hstmt and creates a table in the database.

SQLTables
Lists tables associated with an hdbc. The sample driver has only one table associated with any hdbc; it will always be called "TABLE."

SQLColumns
Returns information about the columns associated with the table attached to the hstmt . The sample driver does this by creating a small in-memory database.

SQLGetData
Retrieves the data for a column in the current row in the in-memory database attached to the hstmt .

SQLGetTypeInfo
Returns information about data types supported by the sample driver.

SQLGetConnectOption
Returns connection options that have been set by a previous call to **SQLSetConnectOption**.

SQLSetConnectOption
Stores connect and statement options for the specified hdbc.

SQLGetStmtOption
Returns statement options that have been set by a previous call to **SQLSetStmtOption**.

SQLSetStmtOption
Stores statement options for the specified hstmt.

Testing and Debugging a Driver
The ODBC developer's kit provides the following tools for driver development:

- Test application
- Sample driver

s6

The ODBC test application allows testers and developers to test ODBC drivers on the Windows platform. The test application supports ad-hoc and automated testing procedures.

The test application is a menu-driven ODBC application written in the C programming language. The application screen is split into two pieces; the top half is for input and the bottom half is for output. You can select an ODBC function from a pull down menu, fill in the necessary parameters, and execute the function. You can enter SQL queries into the input half of the screen.

The test application displays the function name, arguments, return codes, and any errors on the output portion of the screen. If you request a query function, the test application displays query results on the output portion of the screen.

To run automated tests, select one or all test cases from the Auto menu. Results from automated tests can be stored in an output file.

For additional information about the test application, refer to the *ODBC Test Application User's Guide*.

s6

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.

n 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**. 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 ::=
ALTER TABLE *base-table-name*
{ ADD *column-identifier data-type*
| ADD (*column-identifier data-type* [, *column-identifier data-type*]...)
}
X

create-index-statement ::=
CREATE [UNIQUE] INDEX *index-name*
ON *base-table-name*
(*column-identifier* [ASC | DESC]
[, *column-identifier* [ASC | DESC]]...)
X

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 ::=
DROP INDEX *index-name*
X

drop-table-statement ::=
DROP TABLE *base-table-name*
[{ CASCADE | RESTRICT }]
X

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

| UPDATE

| REFERENCES

X

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*.
        | .*unsigned-integer* }
X
X

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

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

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

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