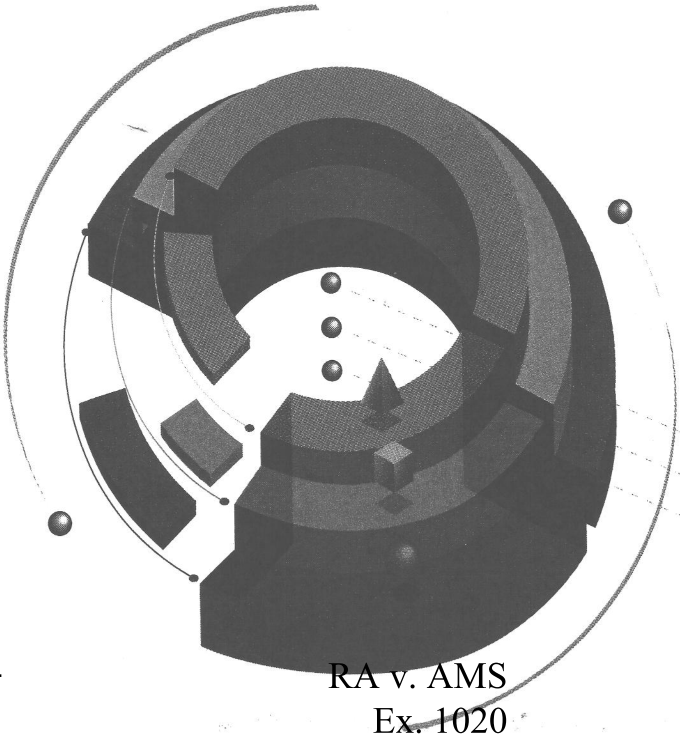
MICROSOFT PROFESSIONAL EDITIONS

MICROSOFT® WINDOWS™

Microsoft® ODBC 2.0 Programmer's Reference and SDK Guide

The Microsoft Open Database Standard for Microsoft Windows™ and Windows NT™



Microsoft PRESS

Microsoft®

ODBC 2.0 Programmer's Reference and SDK Guide

For Microsoft Windows™ and Windows NT™

You'll find detailed coverage of these features, new with version 2.0:

- 32-bit application development support on Windows 3.1 and Windows NT
- Support for scrollable cursors through a driver-independent Cursor Library
- An improved set of ODBC 1.0 single-tier drivers for several popular data formats
- Sample C++ classes to help developers write C++ classes or applications
- Templates that provide a ready-to-use base for writing drivers and auto-test DLLs
- Numerous code samples and working tools to help developers write ODBC-enabled applications
- Improved and integrated installation procedure for the SDK and drivers

U.S.A.	**$24.95**
U.K.	£21.95
Canada	$32.95

[Recommended]

The Microsoft Open Database Connectivity (ODBC) interface is an emerging industry standard and a component of Microsoft's Windows Open Services Architecture (WOSA). A C language programming interface, the ODBC interface enables applications to access data from a variety of database management systems using Structured Query Language (SQL) as a standard. This capability allows for maximum interoperability, making it possible for a developer to develop, compile, and ship an application without tying it to a specific database management system. Users can then add modules called database drivers, which link the application to their choice of database management systems.

The Microsoft ODBC Software Development Kit (SDK), version 2.0, is a set of software components and tools designed to help you develop ODBC drivers and ODBC-enabled applications for the Windows 3.1 and Windows NT operating systems. This volume contains both the ODBC Programmer's Reference and the SDK Guide—the most complete, accurate, and up-to-date information on Microsoft ODBC available anywhere.

In addition to providing the complete ODBC API reference, the Programmer's Reference provides an introduction to ODBC and detailed information about developing applications, developing drivers, and installing and configuring ODBC software. The ODBC SDK Guide offers the technical information you need to install the SDK; manage data sources and drivers; and use ODBC tools such as the ODBC Test Interface, ODBC Spy, and the Driver Setup Toolkit.

Microsoft PRESS ®

Microsoft Professional Editions are distributed by Microsoft Press.

ISBN 1-55615-658-8

90000

9 781556 156588

Programming/C/ Data Access/SQL

Programmer's Reference

Microsoft® Open Database Connectivity™ Software Development Kit

Version 2.0

For the Microsoft Windows™ and Windows NT™ Operating Systems

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Microsoft ODBC 2.0 programmer's reference and SDK guide : for
  Microsoft Windows and Windows NT / Microsoft Corporation.
    p.   cm.
    Includes index.
    ISBN 1-55615-658-8
    1. Microsoft Windows (Computer file)   2. Windows NT.   3. ODBC.
  I. Microsoft Corporation.
  QA76.76.W56M56323   1994
  005.75'8--dc20   94-5039
                                                      CIP

Lucida Typeface Software. © 1985–1988 and 1990 by Bigelow & Holmes.
U.S. Patent Nos. D289420, D289421, D289422, D289773

U. S. Patent No. 4955066

Printed and bound in the United States of America.

2 3 4 5 6 7 8 9 MLML 9 8 7 6 5 4

Contents

Part 2 Developing Applications

Part 5 API Reference

About This Manual

The Microsoft® Open Database Connectivity™ (ODBC) interface is a C programming language interface for database connectivity. This manual addresses the following questions:

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

The following topics provide information about the organization of this manual, describe the knowledge necessary to use the ODBC interface effectively, set out the typographic conventions used, and give a listing of references that provide information about Structured Query Language (SQL) standards and SQL in conjunction with relational databases.

Organization of this Manual

This manual is organized into the following parts:

- Part 1 *Introduction to ODBC*, providing conceptual information about the ODBC interface and a brief history of Structured Query Language;
- Part 2 *Developing Applications*, containing information for developing applications using the ODBC interface;
- Part 3 *Developing Drivers*, containing information for developing drivers that support ODBC function calls;
- Part 4 *Installing and Configuring ODBC Software*, providing information about installation and a setup DLL function reference;
- Part 5 *API Reference*, containing syntax and semantic information for all ODBC functions.

Audience

The ODBC software development kit is available for use with the C programming language run with the Microsoft Windows™ operating system and the Microsoft Windows NT™ operating system. Use of the ODBC interface spans four areas: SQL statements, ODBC function calls, C programming, and Windows programming. For information about Windows programming, see the Microsoft Windows and Microsoft Windows NT Software Development Kit development tools for building Microsoft Windows applications. This manual assumes:

- A working knowledge of the C programming language.
- General DBMS knowledge and a familiarity with SQL.

Document Conventions

This manual uses the following typographic conventions.

Format	Used for
WIN.INI	Uppercase letters indicate filenames, SQL statements, macro names, and terms used at the operating-system command level.
`RETCODE SQLFetch(hdbc)`	This font is used for sample command lines and program code.
argument	Italicized words indicate information that the user or the application must provide, or word emphasis.
SQLTransact	Bold type indicates that syntax must be typed exactly as shown, including function names.
[ ]	Brackets indicate optional items; if in bold text, brackets must be included in the syntax.
|	A vertical bar separates two mutually exclusive choices in a syntax line.
{ }	Braces delimit a set of mutually exclusive choices in a syntax line; if in bold text, braces must be included in the syntax.
...	An ellipsis indicates that arguments can be repeated several times.
. . .	A column of three dots indicates continuation of previous lines of code.

Where to Find 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 specification (1992).
- Database Language—SQL: ANSI X3H2 and ISO/IEC JTC1/SC21/WG3 9075:1992 (SQL-92).

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).

CHAPTER 1

ODBC Theory of Operation

The Open Database Connectivity (ODBC) interface allows applications to access data in database management systems (DBMS) using Structured Query Language (SQL) as a standard for accessing data.

The interface permits maximum *interoperability*—a single application can access different database management systems. This allows an application developer to develop, compile, and ship an application without targeting a specific DBMS. Users can then add modules called database *drivers* that link the application to their choice of database management systems.

ODBC History

In the traditional database world, *application* has usually meant a program that performed a specific database task with a specific DBMS in mind such as payroll, financial analysis, or inventory management. Such applications have typically been written using embedded SQL. While embedded SQL is efficient and is portable across different hardware and operating system environments, the source code must be recompiled for each new environment.

Embedded SQL is not optimal for applications that need to analyze data stored in databases such as DB2® and Oracle®, and prefer to do so from within a familiar application interface, such as a Microsoft Excel® spreadsheet. Under the traditional approach to database access, one version of Microsoft Excel would have to be precompiled with the IBM® precompiler and another with the Oracle precompiler, clearly a radical departure from simply buying a single packaged product.

ODBC offers a new approach: provide a separate program to extract the database information, and then have a way for applications to import the data. Since there are and probably always will be many viable communication methods, data protocols, and DBMS capabilities, the ODBC solution is to allow different technologies to be used by defining a standard interface. This solution leads to the idea of database drivers—dynamic-link libraries that an application can invoke on demand to gain access to a particular data source through a particular communications method, much like a printer driver running under Windows. ODBC provides the standard interface that allows both application writers and providers of libraries to shuttle data between applications and data sources.

ODBC Interface

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 based on the X/Open and SQL Access Group (SAG) SQL CAE specification (1992).
- A standard set of error codes.
- A standard way to connect and log on to a DBMS.
- A standard representation for data types.

The interface is flexible:

- Strings containing SQL statements can be explicitly included in source code or constructed on the fly at run time.
- The same object code can be used to access different DBMS products.
- An application can ignore underlying data communications protocols between it and a DBMS product.
- Data values can be sent and retrieved in a format convenient to the application.

The ODBC interface provides two types of function calls:

- Core functions are based on the X/Open and SQL Access Group Call Level Interface specification.
- Extended functions support additional functionality, including scrollable cursors and asynchronous processing.

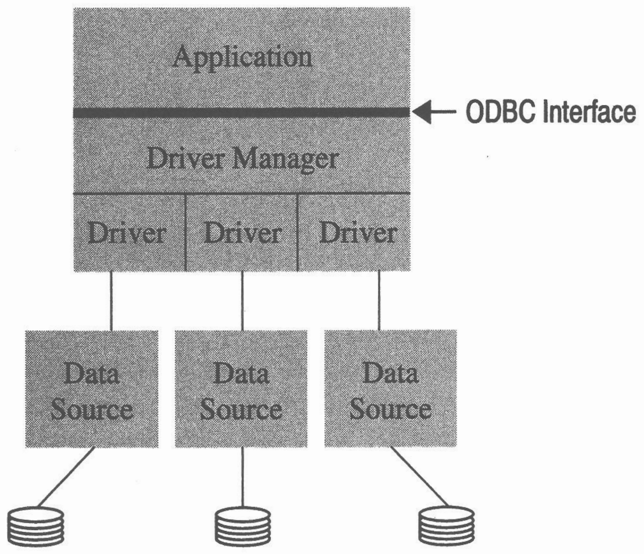
To send an SQL statement, include the statement as an argument in an ODBC function call. The statement need not be customized for a specific DBMS. Appendix C, "SQL Grammar," contains an SQL syntax based on the X/Open and SQL Access Group SQL CAE specification (1992). We recommend that ODBC applications use only the SQL syntax defined in Appendix C to ensure maximum interoperability.

ODBC Components

The ODBC architecture has four components:

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

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



Application

An application using 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 mail, spreadsheet capabilities, online transaction processing, and report generation; the application may or may not interact with users.

Driver Manager

The Driver Manager, provided by Microsoft, is a dynamic-link library (DLL) with an import library. The primary purpose of the Driver Manager is to load drivers. The Driver Manager also performs the following:

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

Driver

A driver is a DLL that implements ODBC function calls and interacts with a data source.

The Driver Manager loads a driver when the application calls the **SQLBrowseConnect**, **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 there is a request for access to a cursor name.)
- Initiates transactions if the data source requires explicit transaction initiation. (This operation is invisible to the application.)

Data Source

In this manual, *DBMS* refers to the general features and functionality provided by an SQL database management system. A *data source* is a specific instance of a combination of a DBMS product and any remote operating system and network necessary to access it.

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

- An Oracle DBMS running on an OS/2® 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

ODBC defines two types of drivers:

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

One system can contain both types of configurations.

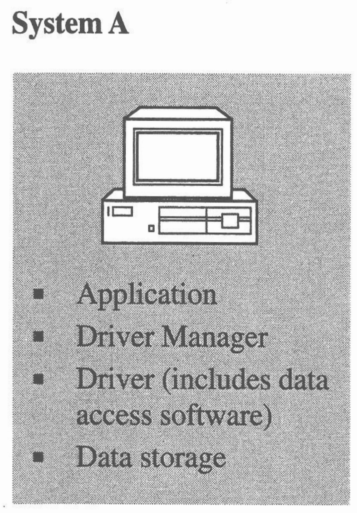
The following paragraphs describe single-tier and multiple-tier configurations in more detail.

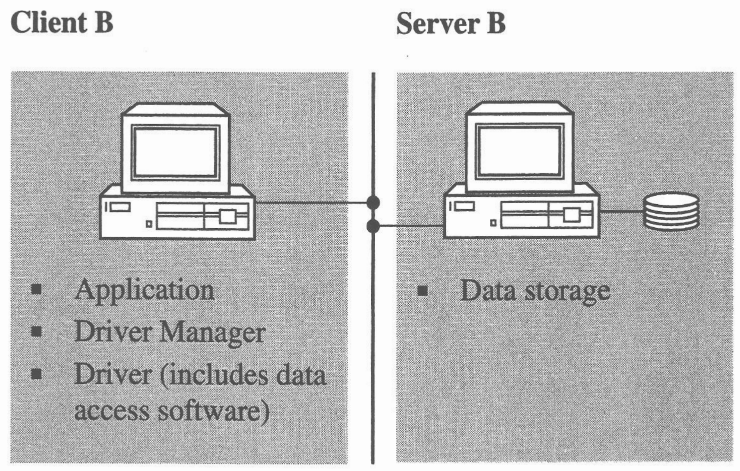
Single-Tier Configuration

In a single-tier implementation, the database file is processed directly by the driver. The driver processes SQL statements and retrieves information from the database. A driver that manipulates an Xbase file is an example of a single-tier implementation.

A single-tier driver may limit the set of SQL statements that may be submitted. The minimum set of SQL statements that must be supported by a single-tier driver is defined in Appendix C, "SQL Grammar."

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





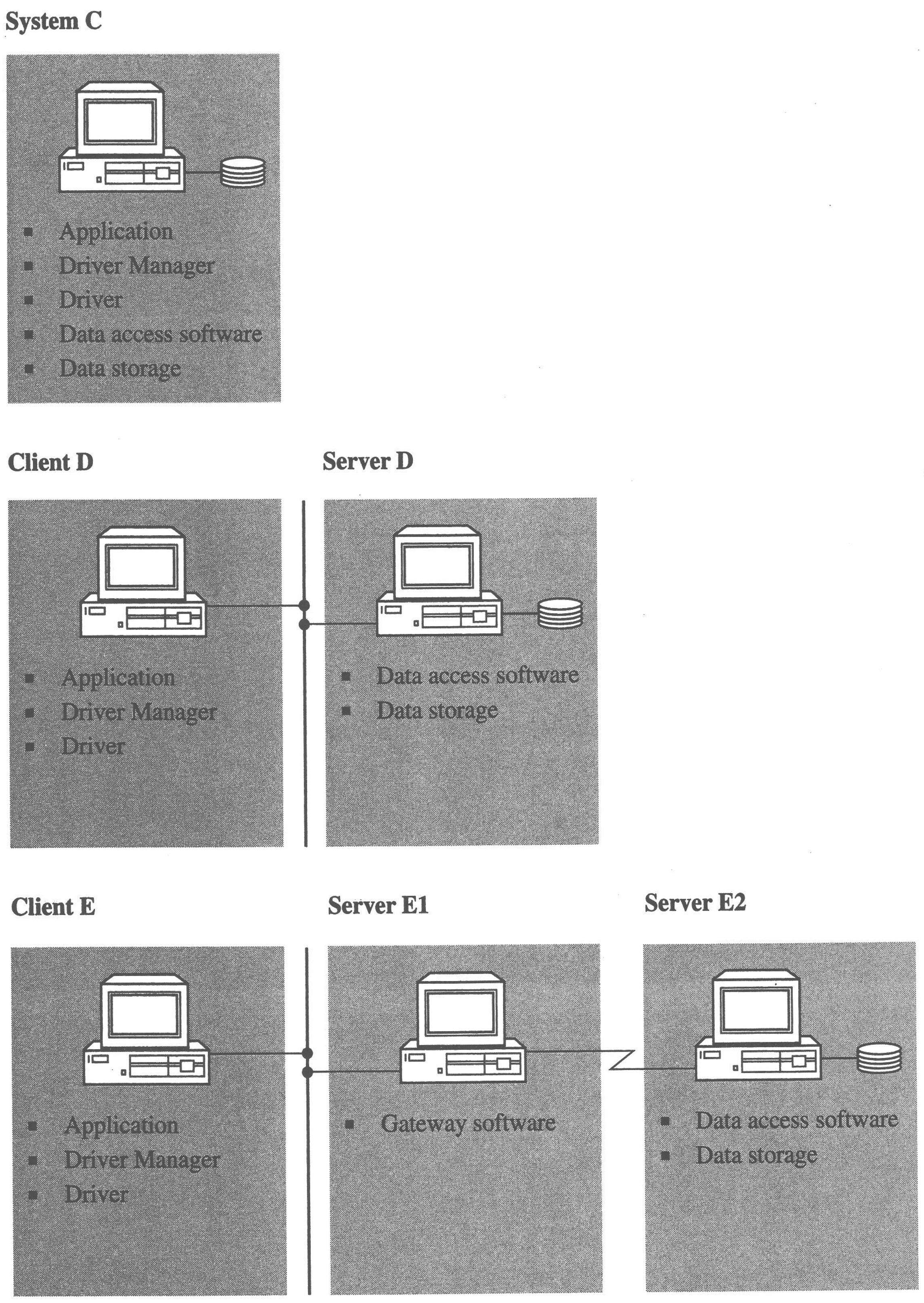
Multiple-Tier Configuration

In a multiple-tier configuration, the driver sends SQL requests to a server that processes SQL requests.

Although the entire installation may reside on a single system, it is more often divided across platforms. The application, driver, and Driver Manager reside on one system, called the client. The database and the software that controls access to the database typically reside on another system, called the server.

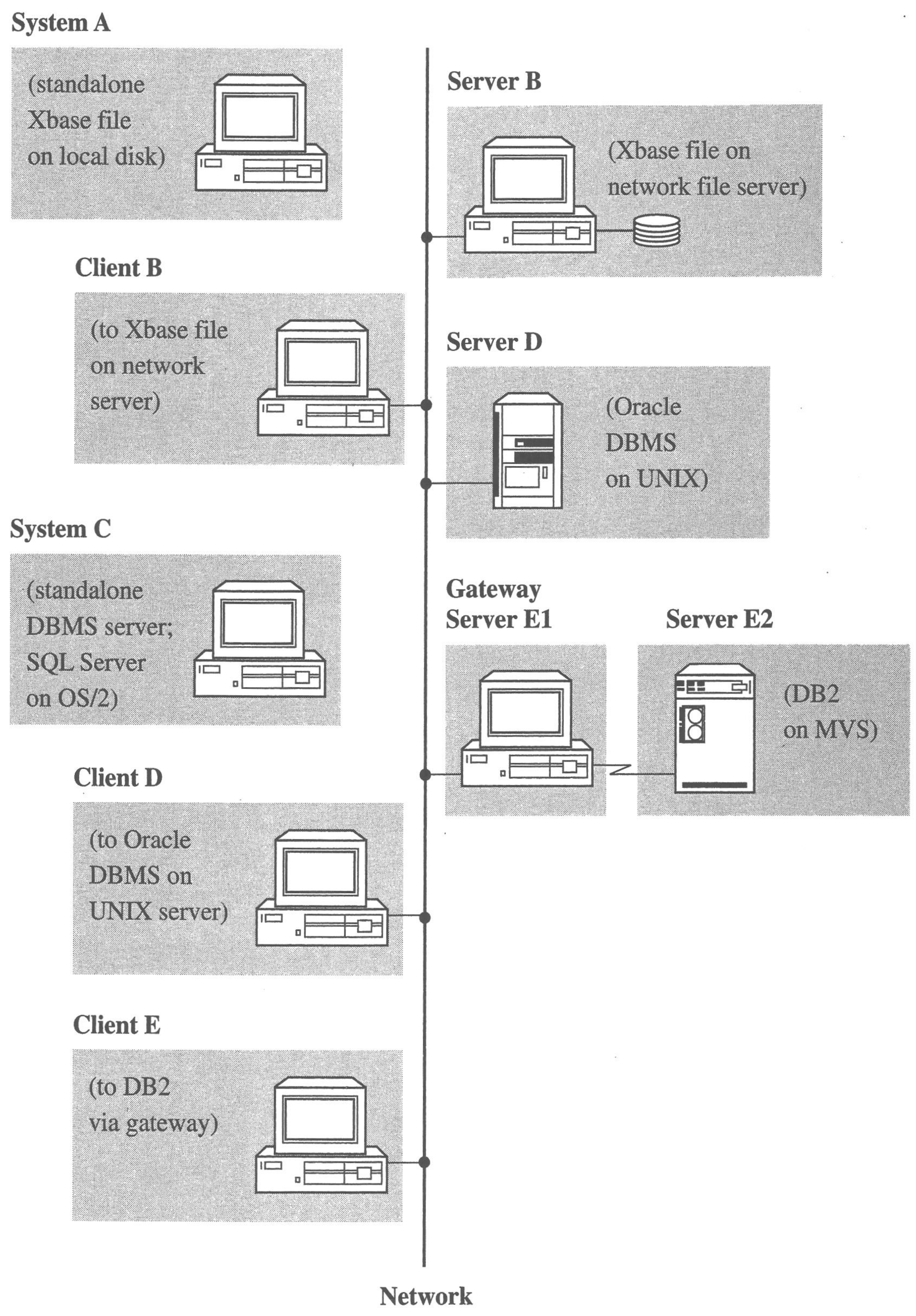
Another type of multiple-tier configuration is a gateway architecture. The driver passes SQL requests to a gateway process, which in turn sends the requests to the data source.

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

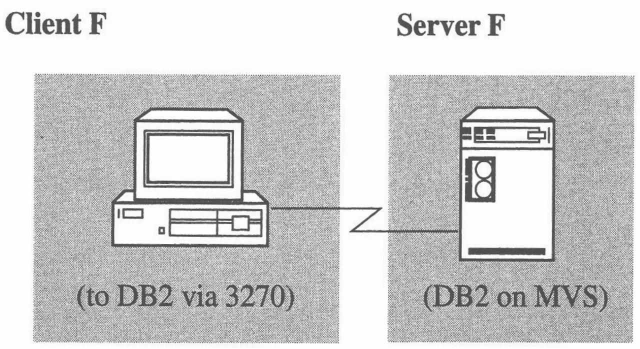


Network Example

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


System A
(standalone Xbase file on local disk)
Server B
(Xbase file on network file server)
Client B
(to Xbase file on network server)
Server D
(Oracle DBMS on UNIX)
System C
(standalone DBMS server; SQL Server on OS/2)
Gateway Server E1
Server E2
(DB2 on MVS)
Client D
(to Oracle DBMS on UNIX server)
Client E
(to DB2 via gateway)
Network

Applications can also communicate across wide area networks:



Matching an Application to a Driver

One of the strengths of the ODBC interface is interoperability; a programmer can create an ODBC application without targeting a specific data source. Users can add drivers to the application after it is compiled and shipped.

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

ODBC Conformance Levels

ODBC defines conformance levels for drivers in two areas: the ODBC API and the ODBC SQL grammar (which includes the ODBC SQL data types). Conformance levels help both application and driver developers by establishing standard sets of functionality. Applications can easily determine if a driver provides the functionality they need. Drivers can be developed to support a broad selection of applications without being concerned about the specific requirements of each application.

To claim that it conforms to a given API or SQL conformance level, a driver must support all the functionality in that conformance level, regardless of whether that functionality is supported by the DBMS associated with the driver. However, conformance levels do not restrict drivers to the functionality in the levels to which they conform. Driver developers are encouraged to support as much functionality as they can; applications can determine the functionality supported by a driver by calling **SQLGetInfo**, **SQLGetFunctions**, and **SQLGetTypeInfo**.

API Conformance Levels

The ODBC API defines a set of core functions that correspond to the functions in the X/Open and SQL Access Group Call Level Interface specification. ODBC also defines two extended sets of functionality, Level 1 and Level 2. The following list summarizes the functionality included in each conformance level.

Important Many ODBC applications require that drivers support all of the functions in the Level 1 API conformance level. To ensure that their driver works with most ODBC applications, driver developers should implement all Level 1 functions.

Core API

- Allocate and free environment, connection, and statement handles.
- Connect to data sources. Use multiple statements on a connection.
- Prepare and execute SQL statements. Execute SQL statements immediately.
- Assign storage for parameters in an SQL statement and result columns.
- Retrieve data from a result set. Retrieve information about a result set.
- Commit or roll back transactions.
- Retrieve error information.

Level 1 API

- Core API functionality.
- Connect to data sources with driver-specific dialog boxes.
- Set and inquire values of statement and connection options.
- Send part or all of a parameter value (useful for long data).
- Retrieve part or all of a result column value (useful for long data).
- Retrieve catalog information (columns, special columns, statistics, and tables).
- Retrieve information about driver and data source capabilities, such as supported data types, scalar functions, and ODBC functions.

Level 2 API

- Core and Level 1 API functionality.
- Browse connection information and list available data sources.
- Send arrays of parameter values. Retrieve arrays of result column values.
- Retrieve the number of parameters and describe individual parameters.
- Use a scrollable cursor.
- Retrieve the native form of an SQL statement.
- Retrieve catalog information (privileges, keys, and procedures).
- Call a translation DLL.

For a list of functions and their conformance levels, see Chapter 21, "Function Summary."

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

SQL Conformance Levels

ODBC defines a core grammar that roughly corresponds to the X/Open and SQL Access Group SQL CAE specification (1992). ODBC also defines a minimum grammar, to meet a basic level of ODBC conformance, and an extended grammar, to provide for common DBMS extensions to SQL. The following list summarizes the grammar included in each conformance level.

Minimum SQL Grammar

- Data Definition Language (DDL): **CREATE TABLE** and **DROP TABLE**.
- Data Manipulation Language (DML): simple **SELECT**, **INSERT**, **UPDATE SEARCHED**, and **DELETE SEARCHED**.
- Expressions: simple (such as **A > B + C**).
- Data types: CHAR, VARCHAR, or LONG VARCHAR.

Core SQL Grammar

- Minimum SQL grammar and data types.
- DDL: **ALTER TABLE**, **CREATE INDEX**, **DROP INDEX**, **CREATE VIEW**, **DROP VIEW**, **GRANT,** and **REVOKE**.
- DML: full **SELECT**.
- Expressions: subquery, set functions such as **SUM** and **MIN**.
- Data types: DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE PRECISION.

Extended SQL Grammar

- Minimum and Core SQL grammar and data types.
- DML: outer joins, positioned **UPDATE**, positioned **DELETE**, **SELECT FOR UPDATE**, and unions.

Note In ODBC 1.0, positioned update, positioned delete, and **SELECT FOR UPDATE** statements and the **UNION** clause were part of the core SQL grammar; in ODBC 2.0, they are part of the extended grammar. Applications that use the SQL conformance level to determine whether these statements are supported also need to check the version number of the driver to correctly interpret the information. In particular, applications that use these features with ODBC 1.0 drivers need to explicitly check for these capabilities in ODBC 2.0 drivers.

- Expressions: scalar functions such as **SUBSTRING** and **ABS**, date, time, and timestamp literals.
- Data types: BIT, TINYINT, BIGINT, BINARY, VARBINARY, LONG VARBINARY, DATE, TIME, TIMESTAMP
- Batch SQL statements.
- Procedure calls.

For more information about SQL statements and conformance levels, see Appendix C, "SQL Grammar." The grammar listed in Appendix C is not intended to restrict the set of statements that an application can submit for execution. Drivers should support data source–specific extensions to the SQL language, although interoperable applications should not rely on those extensions.

For more information about data types, see Appendix D, "Data Types."

How to Select a Set of Functionality

The ODBC functions and SQL statements that a driver supports usually depend on the capabilities of its associated data source. Driver developers are encouraged, however, to implement as many ODBC functions as possible to ensure the widest possible use by applications.

The ODBC functions and SQL statements that an application uses depend on:

- The functionality needed by the application.
- The performance needed by the application.
- The data sources to be accessed by the application and the extent to which the application must be interoperable among these data sources.
- The functionality available in the drivers used by the application.

Because drivers support different levels of functionality, application developers may have to make trade-offs among the factors listed above. For example, an application might display the data in a table. It uses **SQLColumnPrivileges** to determine which columns a user can update and dims those columns the user cannot update. If some of the drivers available to the developer of this application do not support **SQLColumnPrivileges**, the developer can decide to:

- Use all the drivers and not dim any columns. The application behaves the same for all data sources, but has reduced functionality: the user might attempt to update data in a column for which they do not have update privileges. The application returns an error message only when the driver attempts to update the data in the data source.
- Use only those drivers that support **SQLColumnPrivileges**. The application behaves the same for all supported data sources, but has reduced functionality: the application does not support all the drivers.

- Use all the drivers and, for drivers that support **SQLColumnPrivileges**, dim columns the user cannot update. Otherwise, warn the user that they might not have update privileges on all columns. The application behaves differently for different data sources but has increased functionality: the application supports all drivers and sometimes dims columns the user cannot update.
- Use all the drivers and always dim columns the user cannot update; the application locally implements **SQLColumnPrivileges** for those drivers that do not support it. The application behaves the same for all data sources and has maximum functionality. However, the developer must know how to retrieve column privileges from some of the data sources, the application contains data source–specific code, and developement time is longer.

Developers of specialized applications may make different trade-offs than developers of generalized applications. For example, the developer of an application that only transfers data between two DBMS's (each from a different vendor) can safely exploit the full functionality of each of the drivers.

Connections and Transactions

Before an application can use ODBC, it must initialize ODBC and request an environment handle (*henv*). To communicate with a data source, the application must request a connection handle (*hdbc*) and connect to the data source. The application uses the environment and connection handles in subsequent ODBC calls to refer to the environment and specific connection.

An application may 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.

A driver maintains a transaction for each active connection. The application can request that each SQL statement be automatically committed on 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.

The Driver Manager manages the work of allowing an application to switch connections while transactions are in progress on the current connection.

CHAPTER 3

Guidelines for Calling ODBC Functions

This chapter describes the general characteristics of ODBC functions, determining driver conformance levels, the role of the Driver Manager, ODBC function arguments, and the values ODBC functions return.

General Information

Each ODBC function name starts with the prefix "SQL." Each function accepts one or more arguments. Arguments are defined as input (to the driver) or output (from the driver).

C programs that call ODBC functions must include the SQL.H, SQLEXT.H, and WINDOWS.H header files. These files define Windows and ODBC constants and types and provide function prototypes for all ODBC functions.

Determining Driver Conformance Levels

ODBC defines conformance levels for drivers in two areas: the ODBC API and the ODBC SQL grammar (which includes the ODBC SQL data types). These levels establish standard sets of functionality. By inquiring the conformance levels supported by a driver, an application can easily determine if the driver provides the necessary functionality. For a complete discussion of ODBC conformance levels, see "ODBC Conformance Levels" in Chapter 1, "ODBC Theory of Operation."

Note The following sections refer to **SQLGetInfo** and **SQLGetTypeInfo**, which are part of the Level 1 API conformance level. Although it is strongly recommended that drivers support this conformance level, drivers are not required to do so. If these functions are not supported, an application developer must consult the driver documentation to determine its conformance levels.

Determining API Conformance Levels

ODBC functions are divided into core functions, which are defined in the X/Open and SQL Access Group Call Level Interface specification, and two levels of extension functions, with which ODBC extends this specification. To determine the function conformance level of a driver, an application calls **SQLGetInfo** with the SQL_ODBC_SAG_CLI_CONFORMANCE and SQL_ODBC_API_CONFORMANCE flags. Note that a driver can support one or more extension functions but not conform to ODBC extension Level 1 or 2. To determine if a driver supports a particular function, an application calls **SQLGetFunctions**. Note that **SQLGetFunctions** is implemented by the Driver Manager and can be called for any driver, regardless of its level.

Determining SQL Conformance Levels

The ODBC SQL grammar, which includes SQL data types, is divided into a minimum grammar, a core grammar, which corresponds to the X/Open and SQL Access Group SQL CAE specification (1992), and an extended grammar, which provides common extensions to SQL. To determine the SQL conformance level of a driver, an application calls **SQLGetInfo** with the SQL_ODBC_SQL_CONFORMANCE flag. To determine whether a driver supports a specific SQL extension, an application calls **SQLGetInfo** with a flag for that extension. For more information, see Appendix C, "SQL Grammar." To determine whether a driver supports a specific SQL data type, an application calls **SQLGetTypeInfo**.

Using the Driver Manager

The Driver Manager is a DLL that provides access to ODBC drivers. An application typically links with the Driver Manager import library (ODBC.LIB) to gain access to the Driver Manager.

Whenever an application calls an ODBC function, the Driver Manager performs one of the following actions:

- For **SQLDataSources** and **SQLDrivers**, the Driver Manager processes the call. It does not pass the call to the driver.
- For **SQLGetFunctions**, the Driver Manager passes the call to the driver associated with the connection. If the driver does not support **SQLGetFunctions**, the Driver Manager processes the call.
- For **SQLAllocEnv**, **SQLAllocConnect**, **SQLSetConnectOption**, **SQLFreeConnect**, and **SQLFreeEnv**, the Driver Manager processes the call. The Driver Manager calls **SQLAllocEnv**, **SQLAllocConnect**, and **SQLSetConnectOption** in the driver when the application calls a function to

connect to the data source (**SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**). The Driver Manager calls **SQLFreeConnect** and **SQLFreeEnv** in the driver when the application calls **SQLDisconnect**.

- For **SQLConnect**, **SQLDriverConnect**, **SQLBrowseConnect**, and **SQLError**, the Driver Manager performs initial processing then passes 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.

If requested, the Driver Manager records each called function in a trace file. The name of each function is recorded, along with the values of the input arguments and the names of the output arguments (as listed in the function definitions).

Calling ODBC Functions

The following paragraphs describe general characteristics of ODBC functions.

Buffers

An application passes data to a driver in an input buffer. The driver returns data to an application in an output buffer. The application must allocate memory for both input and output buffers. (If the application will use the buffer to retrieve string data, the buffer must contain space for the null termination byte.)

Note that some functions accept pointers to buffers that are later used by other functions. The application must ensure that these pointers remain valid until all applicable functions have used them. For example, the argument *rgbValue* in **SQLBindCol** points to an output buffer in which **SQLFetch** returns the data for a column.

Caution ODBC does not require drivers to correctly manage buffers that cross segment boundaries in Windows 3.1. The Driver Manager supports the use of such buffers, since it passes buffer addresses to drivers and does not operate on buffer contents. If a driver supports buffers that cross segment boundaries, the documentation for the driver should clearly state this.

For maximum interoperability, applications that use buffers that cross segment boundaries should pass them in pieces to ODBC functions. None of these pieces can cross a segment boundary. For example, suppose a data source contains 100 kilobytes of bitmap data. A Windows 3.1 application can safely allocate 100K of memory (beginning at a segment boundary) and retrieve the data in two pieces (64K and 36K), each of which begins on a segment boundary.

Input Buffers

An application passes the address and length of an input buffer to a driver. The length of the buffer must be one of the following values:

- A length greater than or equal to zero. This is the actual length of the data in the input buffer. For character data, a length of zero indicates that the data is an empty (zero length) string. Note that this is different from a null pointer. If the application specifies the length of character data, the character data does not need to be null-terminated.
- SQL_NTS. This specifies that a character data value is null-terminated.
- SQL_NULL_DATA. This tells the driver to ignore the value in the input buffer and use a NULL data value instead. It is only valid when the input buffer is used to provide the value of a parameter in an SQL statement.

The operation of ODBC functions on character data containing embedded null characters is undefined, and is not recommended for maximum interoperability.

Unless it is specifically prohibited in a function description, the address of an input buffer may be a null pointer. When the address of an input buffer is a null pointer, the value of the corresponding buffer length argument is ignored.

For more information about input buffers, see "Converting Data from C to SQL Data Types" in Appendix D, "Data Types."

Output Buffers

An application passes the following arguments to a driver, so that it can return data in an output buffer:

- The address of the buffer in which the driver returns the data (the output buffer). Unless it is specifically prohibited in a function description, the address of an output buffer can be a null pointer. In this case, the driver does not return anything in the buffer and, in the absence of other errors, returns SQL_SUCCESS.

  If necessary, the driver converts data before returning it. The driver always null-terminates character data before returning it.
- The length of the buffer. This is ignored by the driver if the returned data has a fixed length in C, such as an integer, real number, or date structure.
- The address of a variable in which the driver returns the length of the data (the length buffer). The returned length of the data is SQL_NULL_DATA if the data is a NULL value in a result set. Otherwise, it is the number of bytes of data available to return. If the driver converts the data, it is the number of bytes after the conversion. For character data, it does not include the null termination byte added by the driver.

If the output buffer is too small, the driver attempts to truncate the data. If the truncation does not cause a loss of significant data, the driver returns the truncated data in the output buffer, returns the length of the available data (as opposed to the length of the truncated data) in the length buffer, and returns SQL_SUCCESS_WITH_INFO. If the truncation causes a loss of significant data, the driver leaves the output and length buffers untouched and returns SQL_ERROR. The application calls **SQLError** to retrieve information about the truncation or the error.

For more information about output buffers, see "Converting Data from SQL to C Data Types" in Appendix D, "Data Types."

Environment, Connection, and Statement Handles

When so requested by an application, the Driver Manager and each driver allocate storage for information about the ODBC environment, each connection, and each SQL statement. The handles to these storage areas are returned to the application. The application then uses one or more of them in each call to an ODBC function.

The ODBC interface defines three types of handles:

- The **environment handle** identifies memory storage for global information, including the valid connection handles and the current active connection handle. ODBC defines the environment handle as a variable of type HENV. An application uses a single environment handle; it must request this 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 connecting to a data source. Each connection handle is associated with the environment handle. The environment handle can, however, have multiple connection handles associated with it.
- **Statement handles** identify memory storage for information about an SQL statement. ODBC defines statement handles as variables of type HSTMT. An application must request a statement handle prior to submitting SQL requests. Each statement handle is associated with exactly one connection handle. Each connection handle can, however, have multiple statement handles associated with it.

For more information about requesting a connection handle, see Chapter 5, "Connecting to a Data Source." For more information about requesting a statement handle, see Chapter 6, "Executing SQL Statements."

Using Data Types

Data stored on a data source has an SQL data type, which may be specific to that data source. A driver maps data source–specific SQL data types to ODBC SQL data types, which are defined in the ODBC SQL grammar, and driver-specific SQL data types. (A driver returns these mappings through **SQLGetTypeInfo**. It also uses the ODBC SQL data types to describe the data types of columns and parameters in **SQLColAttributes**, **SQLDescribeCol**, and **SQLDescribeParam**.)

Each SQL data type corresponds to an ODBC C data type. By default, the driver assumes that the C data type of a storage location corresponds to the SQL data type of the column or parameter to which the location is bound. If the C data type of a storage location is not the *default* C data type, the application can specify the correct C data type with the *fCType* argument in **SQLBindCol**, **SQLGetData**, or **SQLBindParameter**. Before returning data from the data source, the driver converts it to the specified C data type. Before sending data to the data source, the driver converts it from the specified C data type.

For more information about data types, see Appendix D, "Data Types." The C data types are defined in SQL.H and SQLEXT.H.

ODBC Function Return Codes

When an application calls an ODBC function, the driver executes the function and returns a predefined code. These return codes indicate success, warning, or failure status. The return codes are:

SQL_SUCCESS
SQL_SUCCESS_WITH_INFO
SQL_NO_DATA_FOUND
SQL_ERROR
SQL_INVALID_HANDLE
SQL_STILL_EXECUTING
SQL_NEED_DATA

If the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, the application can call **SQLError** to retrieve additional information about the error. For a complete description of return codes and error handling, see Chapter 8, "Retrieving Status and Error Information."

CHAPTER 5

Connecting to a Data Source

This chapter briefly describes data sources. It then describes how to establish a connection to a data source.

About Data Sources

A data source consists of the data a user wants to access, its associated DBMS, the platform on which the DBMS resides, and the network (if any) used to access that platform. Each data source requires that a driver provide certain information in order to connect to it. At the core level, this is defined to be the name of the data source, a user ID, and a password. ODBC extensions allow drivers to specify additional information, such as a network address or additional passwords.

The connection information for each data source is stored in the ODBC.INI file or registry, which is created during installation and maintained with an administration program. A section in this file lists the available data sources. Additional sections describe each data source in detail, specifying the driver name, a description, and any additional information the driver needs to connect to the data source.

For example, suppose a user has three data sources: Personnel and Inventory, which use an Rdb DBMS, and Payroll, which uses an SQL Server DBMS. The section that lists the data sources might be:

```
[ODBC Data Sources]
Personnel=Rdb
Inventory=Rdb
Payroll=SQL Server
```

Suppose also that an Rdb driver needs the ID of the last user to log in, a server name, and a schema declaration statement. The section that describes the Personnel data source might be:

```
[Personnel]
Driver=c:\windows\system\rdb.dll
Description=Personnel database: CURLY
Lastuid=smithjo
Server=curly
Schema=declare schema personnel filename
➥ "sys$sysdevice:[corpdata]personnel.rdb"
```

For more information about data sources and how to configure them, see Chapter 20, "Configuring Data Sources."

Initializing the ODBC Environment

Before an application can use any other ODBC function, it must initialize the ODBC interface and associate an environment handle with the environment. To initialize the interface and allocate an environment handle, an application:

1. Declares a variable of type HENV. For example, the application could use the declaration:

   ```
   HENV henv1;
   ```

2. Calls **SQLAllocEnv** and passes it the address of the variable. The driver initializes the ODBC environment, allocates memory to store information about the environment, and returns the environment handle in the variable.

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

Allocating a Connection Handle

Before an application can connect to a driver, it must allocate a connection handle for the connection. To allocate a connection handle, an application:

1. Declares a variable of type HDBC. For example, the application could use the declaration:

   ```
   HDBC hdbc1;
   ```

2. Calls **SQLAllocConnect** and passes it the address of the variable. The driver allocates memory to store information about the connection and returns the connection handle in the variable.

Connecting to a Data Source

Next, the application specifies a specific driver and data source. It passes the following information to the driver in a call to **SQLConnect**:

- **Data source name** The name of the data source being requested by the application.
- **User ID** The login ID or account name for access to the data source, if appropriate (optional).
- **Authentication string (password)** A character string associated with the user ID that allows access to the data source (optional).

When an application calls **SQLConnect**, the Driver Manager uses the data source name to read the name of the driver DLL from the appropriate section of the ODBC.INI file or registry. It then loads the driver DLL and passes the **SQLConnect** arguments to it. If the driver needs additional information to connect to the data source, it reads this information from the same section of the ODBC.INI file.

If the application specifies a data source name that is not in the ODBC.INI file or registry, or if the application does not specify a data source name, the Driver Manager searches for the default data source specification. If it finds the default data source, it loads the default driver DLL and passes the application-specified data source name to it. If there is no default data source, the Driver Manager returns an error.

ODBC Extensions for Connections

ODBC extends the X/Open and SQL Access Group Call Level Interface to provide additional functions related to connections, drivers, and data sources. The remainder of this chapter describes these functions. To determine if a driver supports a specific function, an application calls **SQLGetFunctions**.

Connecting to a Data Source With SQLDriverConnect

SQLDriverConnect supports:

- Data sources that require more connection information than the three arguments in **SQLConnect**.
- Dialog boxes to prompt the user for all connection information.
- Data sources that are not defined in the ODBC.INI file or registry.

SQLDriverConnect uses a connection string to specify the information needed to connect to a driver and data source.

A connection string contains the following information:

- Data source name or driver description
- Zero or more user IDs
- Zero or more passwords
- Zero or more data source–specific parameter values

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

An application calls **SQLDriverConnect** in one of three ways:

- Specifies a connection string that contains a data source name. The Driver Manager retrieves the full path of the driver DLL associated with the data source from the ODBC.INI file or registry. To retrieve a list of data source names, an application calls **SQLDataSources**.
- Specifies a connection string that contains a driver description. The Driver Manager retrieves the full path of the driver DLL. To retrieve a list of driver descriptions, an application calls **SQLDrivers**.
- Specifies a connection string that does not contain a data source name or a driver description. The Driver Manager displays a dialog box from which the user selects a data source name. The Driver Manager then retrieves the full path of the driver DLL associated with the data source.

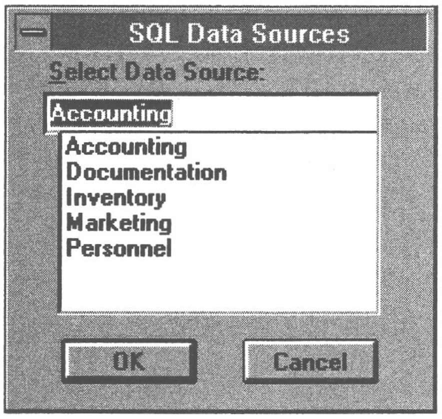
The Driver Manager then loads the driver DLL and passes the **SQLDriverConnect** arguments to it.

The application may pass all the connection information the driver needs. It may also request that the driver always prompt the user for connection information or only prompt the user for information it needs. Finally, if a data source is specified, the driver may read connection information from the appropriate section of the ODBC.INI file or registry. (For information on the structure of the ODBC.INI file or the subkeys used in the registry, see "Structure of the ODBC.INI File" in Chapter 20, "Configuring Data Sources.")

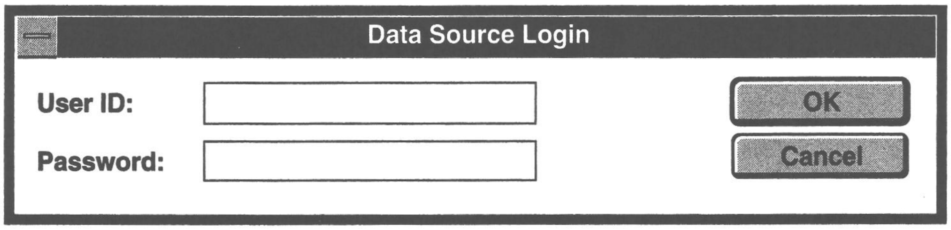
After the driver connects to the data source, it returns the connection information to the application. The application may store this information for future use.

If the application specifies a data source name that is not in the ODBC.INI file or registry, the Driver Manager searches for the default data source specification. If it finds the default data source, it loads the default driver DLL and passes the application-specified data source name to it. If there is no default data source, the Driver Manager returns an error.

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



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



Connection Browsing With SQLBrowseConnect

SQLBrowseConnect supports an iterative method of listing and specifying the attributes and attribute values required to connect to a data source. For each level of a connection, an application calls **SQLBrowseConnect** and specifies the connection attributes and attribute values for that level. First level connection attributes always include the data source name or driver description; the connection attributes for later levels are data source–dependent, but might include the host, user name, and database.

Each time **SQLBrowseConnect** is called, it validates the current attributes, returns the next level of attributes, and returns a user-friendly name for each attribute. It may also return a list of valid values for those attributes. (Note, however, that for some drivers and attributes, this list may not be complete.) After an application has specified each level of attributes and values, **SQLBrowseConnect** connects to the data source and returns a complete connection string. This string can be used in conjunction with **SQLDriverConnect** to connect to the data source at a later time.

Connection Browsing Example for SQL Server

The following example shows how **SQLBrowseConnect** might be used to browse the connections available with a driver for Microsoft's SQL Server. Although other drivers may require different connection attributes, this example illustrates the connection browsing model. (For the syntax of browse request and result strings, see **SQLBrowseConnect** in Chapter 22, "ODBC Function Reference.")

First, the application requests a connection handle:

```
SQLAllocConnect(henv, &hdbc);
```

Next, the application calls **SQLBrowseConnect** and specifies a data source name:

```
SQLBrowseConnect(hdbc, "DSN=MySQLServer", SQL_NTS,
                 szBrowseResult, 100, &cb);
```

Because this is the first call to **SQLBrowseConnect**, the Driver Manager locates the data source name (MySQLServer) in the ODBC.INI file and loads the corresponding driver DLL (SQLSRVR.DLL). The Driver Manager then calls the driver's **SQLBrowseConnect** function with the same arguments it received from the application.

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

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

Note that each keyword in the browse result string is followed by a colon and one or more words before the equal sign. These words are the user-friendly name that an application can use as a prompt in a dialog box.

In its next call to **SQLBrowseConnect**, the application must supply a value for the **SERVER**, **UID**, and **PWD** keywords. Because they are prefixed by an asterisk, the **APP** and **WSID** keywords are optional and may be omitted. The value for the **SERVER** keyword may be one of the servers returned by **SQLBrowseConnect** or a user-supplied name.

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

```
SQLBrowseConnect(hdbc, "SERVER=green;UID=Smith;PWD=Sesame", SQL_NTS,
                 szBrowseResult, 100, &cb);
```

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

```
"*DATABASE:Database={master,model,pubs,tempdb};
➥ *LANGUAGE:Language={us_english,Français}"
```

Since the attributes in this string are optional, the application can omit them. However, the application must call **SQLBrowseConnect** again. If the application chooses to omit the database name and language, it specifies an empty browse request string. In this example, the application chooses the pubs database and calls **SQLBrowseConnect** a final time, omitting the **LANGUAGE** keyword and the asterisk before the **DATABASE** keyword:

```
SQLBrowseConnect(hdbc, "DATABASE=pubs", SQL_NTS,
                 szBrowseResult, 100, &cb);
```

Since the **DATABASE** attribute is the final connection attribute of the data source, the browsing process is complete, the application is connected to the data source, and **SQLBrowseConnect** returns SQL_SUCCESS. **SQLBrowseConnect** also returns the complete connection string as the browse result string:

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

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

```
SQLDriverConnect(hdbc, szBrowseResult, SQL_NTS, szConnStrOut, 100, &cb,
                 SQL_DRIVER_NOPROMPT);
```

Connection Browsing Example for DAL

The following example shows how **SQLBrowseConnect** might be used in conjunction with a driver that uses Apple's Data Access Language (DAL) to access an Oracle host. To browse the available connections, an application repeatedly calls **SQLBrowseConnect**:

```
retcode = SQLBrowseConnect(hdbc, szConnStrIn, SQL_NTS,
                           szConnStrOut, 200, &cb);
```

In the first call, the application specifies a data source name in *szConnStrIn*. In each subsequent call, the application bases the value of *szConnStrIn* on the value of *szConnStrOut* returned by the previous call. The application continues to call

SQLBrowseConnect as long as the function returns SQL_NEED_DATA. The following list shows, for each call to **SQLBrowseConnect**, the value that the application specifies for *szConnStrIn* and the values that the driver returns for *retcode* and *szConnStrOut*. (For the syntax of the strings used in *szConnStrIn* and *szConnStrOut*, see **SQLBrowseConnect** in Chapter 22, "ODBC Function Reference.")

```
szConnStrIn : "DSN=DAL"
szConnStrOut: "HOST:Host={MyVax,Direct,Unix};UID1:Host User Name=?;
              ➥ PWD1:Password=?"
retcode     : SQL_NEED_DATA

szConnStrIn : "HOST=MyVax;UID1=Smith;PWD1=Sesame"
szConnStrOut: "DBMS:DBMS={Oracle,Informix,Sybase};UID2:DBMS User Name=?;
              ➥ PWD2:Password=?"
retcode     : SQL_NEED_DATA

szConnStrIn : "DBMS=Oracle;UID2=John;PWD2=Lion"
szConnStrOut: "DATABASE:Database={DalDemo,Personnel,Production};
              ➥ *ALIAS:Alias=?;*UID3:User Name=?;*PWD3:Password=?"
retcode     : SQL_NEED_DATA

szConnStrIn : "DATABASE=DalDemo;ALIAS=Demo"
szConnStrOut: "DSN=DAL;HOST=MyVax;UID1=Smith;PWD1=Sesame;DBMS=Oracle;
              ➥ UID2=John;PWD2=Lion;DATABASE=DalDemo;ALIAS=Demo"
retcode     : SQL_SUCCESS
```

Note that the database alias, database user name, and database password are optional, as indicated by the asterisk before those attribute names. The application chooses not to specify the user name and password.

Translating Data

An application and a data source can store data in different formats. For example, the application might use a different character set than the data source. ODBC provides a mechanism by which a driver can translate all data (data values, SQL statements, table names, row counts, and so on) that passes between the driver and the data source.

The driver translates data by calling functions in a translation DLL. A default translation DLL can be specified for the data source in the ODBC.INI file or registry; the application can override this by calling **SQLSetConnectOption**. When the driver connects to the data source, it loads the translation DLL (if one has been specified). After the driver has connected to the data source, the application may specify a new translation DLL by calling **SQLSetConnectOption**. For more information about specifying a default translation DLL, see "Specifying a Default Translator" in Chapter 20, "Configuring Data Sources."

Translation functions may support several different types of translation. For example, a function that translates data from one character set to another might support a variety of character sets. To specify a particular type of translation, an application can pass an option flag to the translation functions with **SQLSetConnectOption**.

Additional Extension Functions

ODBC also provides the following functions related to connections, drivers, and data sources. For more information about these functions, see Chapter 22, "ODBC Function Reference."

Function	Description
SQLDataSources	Retrieves a list of available data sources. The Driver Manager retrieves this information from the ODBC.INI file or registry. An application can present this information to a user or automatically select a data source.
SQLDrivers	Retrieves a list of installed drivers and their attributes. The Driver Manager retrieves this information from the ODBCINST.INI file or registry. An application can present this information to a user or automatically select a driver.
SQLGetFunctions	Retrieves functions supported by a driver. This function allows an application to determine at run time whether a particular function is supported by a driver.
SQLGetInfo	Retrieves general information about a driver and data source, including filenames, versions, conformance levels, and capabilities.
SQLGetTypeInfo	Retrieves the SQL data types supported by a driver and data source.
SQLSetConnectOption **SQLGetConnectOption**	These functions set or retrieve connection options, such as the data source access mode, automatic transaction commitment, timeout values, function tracing, data translation options, and transaction isolation.

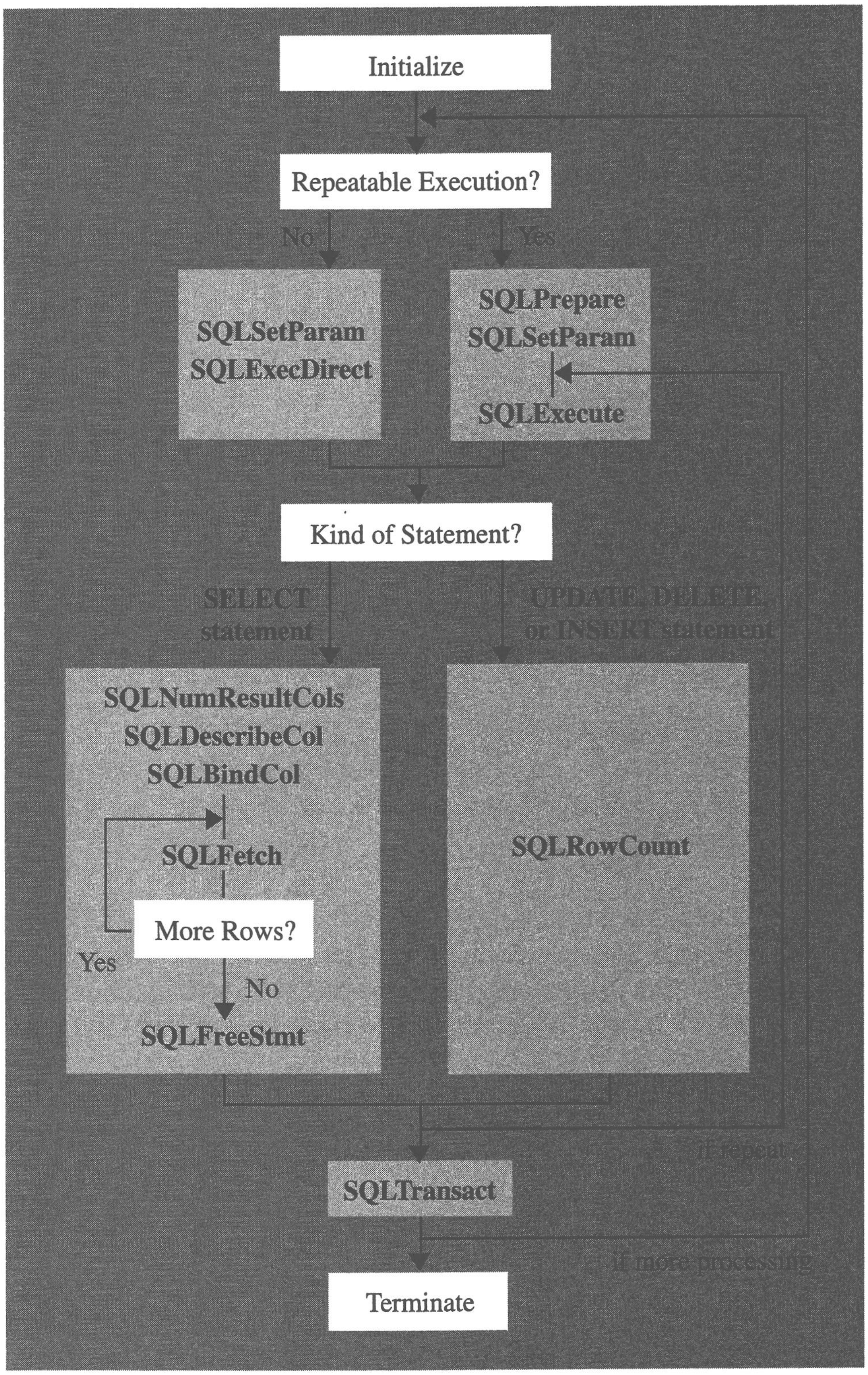
CHAPTER 6

Executing SQL Statements

An application can submit any SQL statement supported by a data source. ODBC defines a standard syntax for SQL statements (listed in Appendix C, "SQL Grammar"). For maximum interoperability, an application should only submit SQL statements that use this syntax; the driver will translate these statements to the syntax used by the data source. If an application submits an SQL statement that does not use the ODBC syntax, the driver passes it directly to the data source.

Note For **CREATE TABLE** and **ALTER TABLE** statements, applications should use the data type name returned by **SQLGetTypeInfo** in the TYPE_NAME column, rather than the data type name defined in the SQL grammar.

The following diagram shows a simple sequence of ODBC function calls to execute SQL statements. Note that statements can be executed a single time with **SQLExecDirect** or prepared with **SQLPrepare** and executed multiple times with **SQLExecute**. Note also that an application calls **SQLTransact** to commit or roll back a transaction.



Allocating a Statement Handle

Before an application can submit an SQL statement, it must allocate a statement handle for the statement. To allocate a statement handle, an application:

1. Declares a variable of type HSTMT. For example, the application could use the declaration:

   ```
   HSTMT hstmt1;
   ```

2. Calls **SQLAllocStmt** and passes it the address of the variable and the connected *hdbc* with which to associate the statement. The driver allocates memory to store information about the statement, associates the statement handle with the *hdbc*, and returns the statement handle in the variable.

Executing an SQL Statement

An application can submit an SQL statement for execution in two ways:

- **Prepared** Call **SQLPrepare** and then call **SQLExecute**.
- **Direct** Call **SQLExecDirect**.

These options are similar, though not identical to, the prepared and immediate options in embedded SQL. For a comparison of the ODBC functions and embedded SQL, see Appendix E, "Comparison Between Embedded SQL and ODBC."

Prepared Execution

An application should prepare a statement before executing it if either of the following is true:

- The application will execute the statement more than once, possibly with intermediate changes to parameter values.
- The application needs information about the result set prior to execution.

A prepared statement executes faster than an unprepared statement because the data source compiles the statement, produces an access plan, and returns an access plan identifier to the driver. The data source minimizes processing time as it does not have to produce an access plan each time it executes the statement. Network traffic is minimized because the driver sends the access plan identifier to the data source instead of the entire statement.

Important Committing or rolling back a transaction, either by calling **SQLTransact** or by using the SQL_AUTOCOMMIT connection option, can cause the data source to delete the access plans for all *hstmts* on an *hdbc*. For more information, see the SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR information types in **SQLGetInfo**.

To prepare and execute an SQL statement, an application:

1. Calls **SQLPrepare** to prepare the statement.
2. Sets the values of any statement parameters. For more information, see "Setting Parameter Values" later in this chapter.
3. Retrieves information about the result set, if necessary. For more information, see "Determining the Characteristics of a Result Set" in Chapter 7, "Retrieving Results."
4. Calls **SQLExecute** to execute the statement.
5. Repeats steps 2 through 4 as necessary.

Direct Execution

An application should execute a statement directly if both of the following are true:

- The application will execute the statement only once.
- The application does not need information about the result set prior to execution.

To execute an SQL statement directly, an application:

1. Sets the values of any statement parameters. For more information, see "Setting Parameter Values" later in this chapter.
2. Calls **SQLExecDirect** to execute the statement.

Setting Parameter Values

An SQL statement can contain parameter markers that indicate values that the driver retrieves from the application at execution time. For example, an application might use the following statement to insert a row of data into the EMPLOYEE table:

```
INSERT INTO EMPLOYEE (NAME, AGE, HIREDATE) VALUES (?, ?, ?)
```

An application uses parameter markers instead of literal values if:

- It needs to execute the same prepared statement several times with different parameter values.
- The parameter values are not known when the statement is prepared.
- The parameter values need to be converted from one data type to another.

To set a parameter value, an application performs the following steps in any order:

- Calls **SQLBindParameter** to bind a storage location to a parameter marker and specify the data types of the storage location and the column associated with the parameter, as well as the precision and scale of the parameter.
- Places the parameter's value in the storage location.

These steps can be performed before or after a statement is prepared, but must be performed before a statement is executed.

Parameter values must be placed in storage locations in the C data types specified in **SQLBindParameter**. For example:

Parameter Value	SQL Data Type	C Data Type	Stored Value
ABC	SQL_CHAR	SQL_C_CHAR	ABC\0 [a]
10	SQL_INTEGER	SQL_C_SLONG	10
10	SQL_INTEGER	SQL_C_CHAR	10\0 [a]
1 P.M.	SQL_TIME	SQL_C_TIME	13,0,0 [b]
1 P.M.	SQL_TIME	SQL_C_CHAR	{t '13:00:00'}\0[a,c]

[a] "\0" represents a null-termination byte; the null termination byte is required only if the parameter length is SQL_NTS.

[b] The numbers in this list are the numbers stored in the fields of the TIME_STRUCT structure.

[c] The string uses the ODBC date escape clause. For more information, see "Date, Time, and Timestamp Data" later in this chapter.

Storage locations remain bound to parameter markers until the application calls **SQLFreeStmt** with the SQL_RESET_PARAMS option or the SQL_DROP option. An application can bind a different storage area to a parameter marker at any time by calling **SQLBindParameter**. An application can also change the value in a storage location at any time. When a statement is executed, the driver uses the current values in the most recently defined storage locations.

Performing Transactions

In *auto-commit* mode, every SQL statement is a complete transaction, which is automatically committed. In *manual-commit* mode, a transaction consists of one or more statements. In manual-commit mode, when an application submits an SQL statement and no transaction is open, the driver implicitly begins a transaction. The transaction remains open until the application commits or rolls back the transaction with **SQLTransact**.

If a driver supports the SQL_AUTOCOMMIT connection option, the default transaction mode is auto-commit; otherwise, it is manual-commit. An application calls **SQLSetConnectOption** to switch between manual-commit and autocommit mode. Note that if an application switches from manual-commit to auto-commit mode, the driver commits any open transactions on the connection.

Applications should call **SQLTransact**, rather than submitting a **COMMIT** or **ROLLBACK** statement, to commit or roll back a transaction. The result of a **COMMIT** or **ROLLBACK** statement depends on the driver and its associated data source.

Important Committing or rolling back a transaction, either by calling **SQLTransact** or by using the SQL_AUTOCOMMIT connection option, can cause the data source to close the cursors and delete the access plans for all *hstmts* on an *hdbc*. For more information, see the SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR information types in **SQLGetInfo**.

ODBC Extensions for SQL Statements

ODBC extends the X/Open and SQL Access Group Call Level Interface to provide additional functions related to SQL statements. ODBC also extends the X/Open and SQL Access Group SQL CAE specification (1992) to provide common extensions to SQL. The remainder of this chapter describes these functions and SQL extensions.

To determine if a driver supports a specific function, an application calls **SQLGetFunctions**. To determine if a driver supports a specific ODBC extension to SQL, such as outer joins or procedure invocation, an application calls **SQLGetInfo**.

Retrieving Information About the Data Source's Catalog

The following functions, known as catalog functions, return information about a data source's catalog:

- **SQLTables** returns the names of tables stored in a data source.
- **SQLTablePrivileges** returns the privileges associated with one or more tables.
- **SQLColumns** returns the names of columns in one or more tables.
- **SQLColumnPrivileges** returns the privileges associated with each column in a single table.
- **SQLPrimaryKeys** returns the names of columns that comprise the primary key of a single table.
- **SQLForeignKeys** returns the names of columns in a single table that are foreign keys. It also returns the names of columns in other tables that refer to the primary key of the specified table.

- **SQLSpecialColumns** returns information about the optimal set of columns that uniquely identify a row in a single table or the columns in that table that are automatically updated when any value in the row is updated by a transaction.
- **SQLStatistics** returns statistics about a single table and the indexes associated with that table.
- **SQLProcedures** returns the names of procedures stored in a data source.
- **SQLProcedureColumns** returns a list of the input and output parameters, as well as the names of columns in the result set, for one or more procedures.

Each function returns the information as a result set. An application retrieves these results by calling **SQLBindCol** and **SQLFetch**.

Sending Parameter Data at Execution Time

To send parameter data at statement execution time, such as for parameters of the SQL_LONGVARCHAR or SQL_LONGVARBINARY types, an application uses the following three functions:

- **SQLBindParameter**
- **SQLParamData**
- **SQLPutData**

To indicate that it plans to send parameter data at statement execution time, an application calls **SQLBindParameter** and sets the *pcbValue* buffer for the parameter to the result of the SQL_LEN_DATA_AT_EXEC(*length*) macro. If the *fSqlType* argument is SQL_LONGVARBINARY or SQL_LONGVARCHAR and the driver returns "Y" for the SQL_NEED_LONG_DATA_LEN information type in **SQLGetInfo**, *length* is the total number of bytes of data to be sent for the parameter; otherwise, it is ignored.

The application sets the *rgbValue* argument to a value that, at run time, can be used to retrieve the data. For example, *rgbValue* might point to a storage location that will contain the data at statement execution time or to a file that contains the data. The driver returns the value to the application at statement execution time.

When the driver processes a call to **SQLExecute** or **SQLExecDirect** and the statement being executed includes a data-at-execution parameter, the driver returns SQL_NEED_DATA. To send the parameter data, the application:

1. Calls **SQLParamData**, which returns *rgbValue* (as set with **SQLBindParameter**) for the first data-at-execution parameter.
2. Calls **SQLPutData** one or more times to send data for the parameter. (More than one call will be needed if the data value is larger than the buffer; multiple

calls are allowed only if the C data type is character or binary and the SQL data type is character, binary, or data source–specific.)

3. Calls **SQLParamData** again to indicate that all data has been sent for the parameter. If there is another data-at-execution parameter, the driver returns *rgbValue* for that parameter and SQL_NEED_DATA for the function return code. Otherwise, it returns SQL_SUCCESS for the function return code.
4. Repeats steps 2 and 3 for the remaining data-at-execution parameters.

For additional information, see the description of **SQLBindParameter** in Chapter 22, "ODBC Function Reference."

Specifying Arrays of Parameter Values

To specify multiple sets of parameter values for a single SQL statement, an application calls **SQLParamOptions**. For example, if there are ten sets of column values to insert into a table—and the same SQL statement can be used for all ten operations—the application can set up an array of values, then submit a single **INSERT** statement.

If an application uses **SQLParamOptions**, it must allocate enough memory to handle the arrays of values.

Executing Functions Asynchronously

By default, a driver executes ODBC functions synchronously; the driver does not return control to an application until a function call completes. If a driver supports asynchronous execution, however, an application can request asynchronous execution for the functions listed below. (All of these functions either submit requests to a data source or retrieve data. These operations may require extensive processing.)

SQLColAttributes	**SQLForeignKeys**	**SQLProcedureColumns**
SQLColumnPrivileges	**SQLGetData**	**SQLProcedures**
SQLColumns	**SQLGetTypeInfo**	**SQLPutData**
SQLDescribeCol	**SQLMoreResults**	**SQLSetPos**
SQLDescribeParam	**SQLNumParams**	**SQLSpecialColumns**
SQLExecDirect	**SQLNumResultCols**	**SQLStatistics**
SQLExecute	**SQLParamData**	**SQLTablePrivileges**
SQLExtendedFetch	**SQLPrepare**	**SQLTables**
SQLFetch	**SQLPrimaryKeys**	

Asynchronous execution is performed on a statement-by-statement basis. To execute a statement asynchronously, an application:

1. Calls **SQLSetStmtOption** with the SQL_ASYNC_ENABLE option to enable asynchronous execution for an *hstmt*. (To enable asynchronous execution for all *hstmts* associated with an *hdbc*, an application calls **SQLSetConnectOption** with the SQL_ASYNC_ENABLE option.)
2. Calls one of the functions listed earlier in this section and passes it the *hstmt*. The driver begins asynchronous execution of the function and returns SQL_STILL_EXECUTING.

   Note If the application calls a function that cannot be executed asynchronously, the driver executes the function synchronously.

3. Performs other operations while the function is executing asynchronously. The application can call any function with a different *hstmt* or an *hdbc* not associated with the original *hstmt*. With the original *hstmt* and the *hdbc* associated with that *hstmt*, the application can only call the original function, **SQLAllocStmt**, **SQLCancel**, or **SQLGetFunctions**.
4. Calls the asynchronously executing function to check if it has finished. While the arguments must be valid, the driver ignores all of them except the *hstmt* argument. For example, suppose an application called **SQLExecDirect** to execute a **SELECT** statement asynchronously. When the application calls **SQLExecDirect** again, the return value indicates the status of the **SELECT** statement, even if the *szSqlStr* argument contains an **INSERT** statement.

   If the function is still executing, the driver returns SQL_STILL_EXECUTING and the application must repeat steps 3 and 4. If the function has finished, the driver returns a different code, such as SQL_SUCCESS or SQL_ERROR. For information about canceling a function executing asynchronously, see "Terminating Statement Processing" in Chapter 9, "Terminating Transactions and Connections."
5. Repeats steps 2 through 4 as needed.

To disable asynchronous execution for an *hstmt*, an application calls **SQLSetStmtOption** with the SQL_ASYNC_ENABLE option. To disable asynchronous execution for all *hstmts* associated with an *hdbc*, an application calls **SQLSetConnectOption** with the SQL_ASYNC_ENABLE option.

Using ODBC Extensions to SQL

ODBC defines the following extensions to SQL, which are common to most DBMS's:

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

- **LIKE** predicate escape characters
- Outer joins
- Procedures

The syntax defined by ODBC for these extensions uses the escape clause provided by the X/Open and SQL Access Group SQL CAE specification (1992) to cover vendor-specific extensions to SQL. Its format is:

--(*vendor(*vendor-name***), product(***product-name***)** *extension* ***)--**

For the ODBC extensions to SQL, *product-name* is always "ODBC", since the product defining them is ODBC. *Vendor-name* is always "Microsoft", since ODBC is a Microsoft product. ODBC also defines a shorthand syntax for these extensions:

{*extension*}

Most DBMS's provide the same extensions to SQL as does ODBC. Because of this, an application may be able to submit an SQL statement using one of these extensions in either of two ways:

- Use the syntax defined by ODBC. An application that uses the ODBC syntax will be interoperable among DBMS's.
- Use the syntax defined by the DBMS. An application that uses DBMS-specific syntax will not be interoperable among DBMS's.

Due to the difficulty in implementing some ODBC extensions to SQL, such as outer joins, a driver might only implement those ODBC extensions that are supported by its associated DBMS. To determine whether the driver and data source support all the ODBC extensions to SQL, an application calls **SQLGetInfo** with the SQL_ODBC_SQL_CONFORMANCE flag. For information about how an application determines whether a specific extension is supported, see the section that describes the extension.

Note Many DBMS's provide extensions to SQL other than those defined by ODBC. To use one of these extensions, an application uses the DBMS-specific syntax. The application will not be interoperable among DBMS's.

Date, Time, and Timestamp Data

The escape clauses ODBC uses for date, time, and timestamp data are:

--(*vendor(Microsoft),product(ODBC) d '*value***' *)--**
--(*vendor(Microsoft),product(ODBC) t '*value***' *)--**
--(*vendor(Microsoft),product(ODBC) ts '*value***' *)--**

where **d** indicates *value* is a date in the "yyyy-mm-dd" format, **t** indicates *value* is a time in the "hh:mm:ss" format, and **ts** indicates *value* is a timestamp in the "yyyy-mm-dd hh:mm:ss[.f...]" format. The shorthand syntax for date, time, and timestamp data is:

{d '*value*'**}**
{t '*value*'**}**
{ts '*value*'**}**

For example, each of the following statements updates the birthday of John Smith in the EMPLOYEE table. The first statement uses the escape clause syntax. The second statement uses the shorthand syntax. The third statement uses the native syntax for a DATE column in DEC's Rdb and is not interoperable among DBMS's.

```
UPDATE EMPLOYEE
    SET BIRTHDAY=--(*vendor(Microsoft),product(ODBC) d '1967-01-15' *)--
    WHERE NAME='Smith, John'

UPDATE EMPLOYEE
    SET BIRTHDAY={d '1967-01-15'}
    WHERE NAME='Smith, John'

UPDATE EMPLOYEE
    SET BIRTHDAY='15-Jan-1967'
    WHERE NAME='Smith, John'
```

The ODBC escape clauses for date, time, and timestamp literals can be used in parameters with a C data type of SQL_C_CHAR. For example, the following statement uses a parameter to update the birthday of John Smith in the EMPLOYEE table:

```
UPDATE EMPLOYEE SET BIRTHDAY=? WHERE NAME='Smith, John'
```

A storage location of type SQL_C_CHAR bound to the parameter might contain any of the following values. The first value uses the escape clause syntax. The second value uses the shorthand syntax. The third value uses the native syntax for a DATE column in DEC's Rdb and is not interoperable among DBMS's.

```
"--(*vendor(Microsoft),product(ODBC) d '1967-01-15' *)--"

"{d '1967-01-15'}"

"'15-Jan-1967'"
```

An application can also send date, time, or timestamp values as parameters using the C structures defined by the C data types SQL_C_DATE, SQL_C_TIME, and SQL_C_TIMESTAMP.

To determine if a data source supports date, time, or timestamp data, an application calls **SQLGetTypeInfo**. If a driver supports date, time, or timestamp data, it must also support the escape clauses for date, time, or timestamp literals.

Scalar Functions

Scalar functions—such as string length, absolute value, or current date—can be used on columns of a result set and on columns that restrict rows of a result set. The escape clause ODBC uses for scalar functions is:

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

where *scalar-function* is one of the functions listed in Appendix F, "Scalar Functions." The shorthand syntax for scalar functions is:

{fn *scalar-function***}**

For example, each of the following statements creates the same result set of uppercase employee names. The first statement uses the escape clause syntax. The second statement uses the shorthand syntax. The third statement uses the native syntax for Ingres™ for OS/2 and is not interoperable among DBMS's.

```
SELECT --(*vendor(Microsoft),product(ODBC) fn UCASE(NAME) *)--
    FROM EMPLOYEE

SELECT {fn UCASE(NAME)} FROM EMPLOYEE

SELECT uppercase(NAME) FROM EMPLOYEE
```

An application can mix scalar functions that use native syntax and scalar functions that use ODBC syntax. For example, the following statement creates a result set of last names of employees in the EMPLOYEE table. (Names in the EMPLOYEE table are stored as a last name, a comma, and a first name.) The statement uses the ODBC scalar function **SUBSTRING** and the SQL Server scalar function **CHARINDEX** and will only execute correctly on SQL Server.

```
SELECT {fn SUBSTRING(NAME, 1, CHARINDEX(',', NAME) - 1)} FROM EMPLOYEE
```

To determine which scalar functions are supported by a data source, an application calls **SQLGetInfo** with the SQL_NUMERIC_FUNCTIONS, SQL_STRING_FUNCTIONS, SQL_SYSTEM_FUNCTIONS, and SQL_TIMEDATE_FUNCTIONS flags.

Data Type Conversion Function

ODBC defines a special scalar function, **CONVERT**, that requests that the data source convert data from one SQL data type to another SQL data type. The escape clause ODBC uses for the **CONVERT** function is:

--(*vendor(Microsoft),product(ODBC)
fn CONVERT(*value_exp***,** *data_type***) *)--**

where *value_exp* is a column name, the result of another scalar function, or a literal value, and *data_type* is a keyword that matches the **#define** name used by an ODBC SQL data type (as defined in Appendix D, "Data Types"). The shorthand syntax for the **CONVERT** function is:

{fn CONVERT(*value_exp***,** *data_type***)}**

For example, the following statement creates a result set of the names and ages of all employees in their twenties. It uses the **CONVERT** function to convert each employee's age from type SQL_SMALLINT to type SQL_CHAR. Each resulting character string is compared to the pattern "2%" to determine if the employee's age is in the twenties.

```
SELECT NAME, AGE FROM EMPLOYEE
    WHERE {fn CONVERT(AGE,SQL_CHAR)} LIKE '2%'
```

To determine if the **CONVERT** function is supported by a data source, an application calls **SQLGetInfo** with the SQL_CONVERT_FUNCTIONS flag. For more information about the **CONVERT** function, see Appendix F, "Scalar Functions."

LIKE Predicate Escape Characters

In a **LIKE** predicate, the percent character (%) matches zero or more of any character and the underscore character (_) matches any one character. The percent and underscore characters can be used as literals in a **LIKE** predicate by preceding them with an escape character. The escape clause ODBC uses to define the **LIKE** predicate escape character is:

--(*vendor(Microsoft),product(ODBC) escape '*escape-character***' *)--**

where *escape-character* is any character supported by the data source. The shorthand syntax for the **LIKE** predicate escape character is:

{escape '*escape-character***'}**

For example, each of the following statements creates the same result set of department names that start with the characters "%AAA". The first statement uses the escape clause syntax. The second statement uses the shorthand syntax. The third statement uses the native syntax for Ingres and is not interoperable among DBMS's. Note that the second percent character in each **LIKE** predicate is a wild-card character that matches zero or more of any character.

```
SELECT NAME FROM DEPT WHERE NAME
    LIKE '\%AAA%' --(*vendor(Microsoft),product(ODBC) escape '\'*)--

SELECT NAME FROM DEPT WHERE NAME LIKE '\%AAA%' {escape '\'}

SELECT NAME FROM DEPT WHERE NAME LIKE '\%AAA%' ESCAPE '\'
```

To determine whether **LIKE** predicate escape characters are supported by a data source, an application calls **SQLGetInfo** with the SQL_LIKE_ESCAPE_CLAUSE information type.

Outer Joins

ODBC supports the ANSI SQL-92 left outer join syntax. The escape clause ODBC uses for outer joins is:

--(*vendor(Microsoft),product(ODBC) oj *outer-join* ***)--**

where *outer-join* is:

table-reference **LEFT OUTER JOIN** {*table-reference* | *outer-join*}
  ON *search-condition*

table-reference specifies a table name, and *search-condition* specifies the join condition between the *table-references*. The shorthand syntax for outer joins is:

{oj *outer-join*}

An outer join request must appear after the **FROM** keyword and before the **WHERE** clause (if one exists). For complete syntax information, see Appendix C, "SQL Grammar."

For example, each of the following statements creates the same result set of the names and departments of employees working on project 544. The first statement uses the escape clause syntax. The second statement uses the shorthand syntax. The third statement uses the native syntax for Oracle and is not interoperable among DBMS's.

```
SELECT EMPLOYEE.NAME, DEPT.DEPTNAME
    FROM --(*vendor(Microsoft),product(ODBC) oj
        EMPLOYEE LEFT OUTER JOIN DEPT ON EMPLOYEE.DEPTID=DEPT.DEPTID*)--
    WHERE EMPLOYEE.PROJID=544

SELECT EMPLOYEE.NAME, DEPT.DEPTNAME
    FROM {oj EMPLOYEE LEFT OUTER JOIN DEPT
        ON EMPLOYEE.DEPTID=DEPT.DEPTID}
    WHERE EMPLOYEE.PROJID=544

SELECT EMPLOYEE.NAME, DEPT.DEPTNAME
    FROM EMPLOYEE, DEPT
    WHERE (EMPLOYEE.PROJID=544) AND (EMPLOYEE.DEPTID = DEPT.DEPTID (+))
```

To determine the level of outer joins a data source supports, an application calls **SQLGetInfo** with the SQL_OUTER_JOINS flag. Data sources can support two-table outer joins, partially support multi-table outer joins, fully support multi-table outer joins, or not support outer joins.

Procedures

An application can call a procedure in place of an SQL statement. The escape clause ODBC uses for calling a procedure is:

--(*vendor(Microsoft),product(ODBC)
[?=] call *procedure-name*[([*parameter*][,[*parameter*]]...)] ***)--**

where *procedure-name* specifies the name of a procedure stored on the data source and *parameter* specifies a procedure parameter. A procedure can have zero or more parameters and can return a value. The shorthand syntax for procedure invocation is:

{[?=]call *procedure-name*[([*parameter*][,[*parameter*]]...)]**}**

For output parameters, *parameter* must be a parameter marker. For input and input/output parameters, *parameter* can be a literal, a parameter marker, or not specified. If *parameter* is a literal or is not specified for an input/output parameter, the driver discards the output value. If *parameter* is not specified for an input or input/output parameter, the procedure uses the default value of the parameter as the input value; the procedure also uses the default value if *parameter* is a parameter marker and the *pcbValue* argument in **SQLBindParameter** is SQL_DEFAULT_PARAM. If a procedure call includes parameter markers (including the "?=" parameter marker for the return value), the application must bind each marker by calling **SQLBindParameter** prior to calling the procedure.

Note For some data sources, *parameter* cannot be a literal value. For all data sources, it can be a parameter marker. For maximum interoperability, applications should always use a parameter marker for *parameter*.

If an application specifies a return value parameter for a procedure that does not return a value, the driver sets the *pcbValue* buffer specified in **SQLBindParameter** for the parameter to SQL_NULL_DATA. If the application omits the return value parameter for a procedure returns a value, the driver ignores the value returned by the procedure.

If a procedure returns a result set, the application retrieves the data in the result set in the same manner as it retrieves data from any other result set.

For example, each of the following statements uses the procedure EMPS_IN_PROJ to create the same result set of names of employees working on a project. The first statement uses the escape clause syntax. The second statement uses the shorthand syntax. For an example of code that calls a procedure, see **SQLProcedures** in Chapter 22, "ODBC Function Reference."

```
--(*vendor(Microsoft),product(ODBC) call EMPS_IN_PROJ(?)*)--

{call EMPS_IN_PROJ(?)}
```

To determine if a data source supports procedures, an application calls **SQLGetInfo** with the SQL_PROCEDURES information type. To retrieve a list of the procedures stored in a data source, an application calls **SQLProcedures**. To retrieve a list of the input, input/output, and output parameters, as well as the return value and the columns that make up the result set (if any) returned by a procedure, an application calls **SQLProcedureColumns**.

Additional Extension Functions

ODBC also provides the following functions related to SQL statements. For more information about these functions, see Chapter 22, "ODBC Function Reference."

Function	Description
SQLDescribeParam	Retrieves information about prepared parameters.
SQLNativeSql	Retrieves the SQL statement as processed by the data source, with escape sequences translated to SQL code used by the data source.
SQLNumParams	Retrieves the number of parameters in an SQL statement.
SQLSetStmtOption **SQLSetConnectOption** **SQLGetStmtOption**	These functions set or retrieve statement options, such as asynchronous processing, orientation for binding rowsets, maximum amount of variable length data to return, maximum number of result set rows to return, and query timeout value. Note that **SQLSetConnectOption** sets options for all statements in a connection.

CHAPTER 11

Guidelines for Implementing ODBC Functions

Each driver supports a set of ODBC functions. These functions perform tasks such as allocating and deallocating memory, transmitting or processing SQL statements, and returning results and errors.

This chapter describes the role of the Driver Manager, the general characteristics of ODBC functions, supporting ODBC conformance levels, ODBC function arguments, and what ODBC functions return.

Role of the Driver Manager

ODBC function calls are passed through the Driver Manager to the driver. An application typically links with the Driver Manager import library (ODBC.LIB) to gain access to the Driver Manager. When an application calls an ODBC function, the Driver Manager performs one of the following actions:

- For **SQLDataSources** and **SQLDrivers**, the Driver Manager processes the call. It does not pass the call to the driver.
- For **SQLGetFunctions**, the Driver Manager passes the call to the driver associated with the connection. If the driver does not support **SQLGetFunctions**, the Driver Manager processes the call.
- For **SQLAllocEnv**, **SQLAllocConnect**, **SQLSetConnectOption**, **SQLFreeConnect**, and **SQLFreeEnv**, the Driver Manager processes the call. The Driver Manager calls **SQLAllocEnv**, **SQLAllocConnect**, and **SQLSetConnectOption** in the driver when the application calls a function to connect to the data source (**SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**). The Driver Manager calls **SQLFreeConnect** and **SQLFreeEnv** in the driver when the application calls **SQLFreeConnect**.
- For **SQLConnect**, **SQLDriverConnect**, **SQLBrowseConnect**, and **SQLError**, 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.

If requested, the Driver Manager records each called function in a trace file after checking the function call for errors. The name of each function that does not contain errors detectable by the Driver Manager is recorded, along with the values of the input arguments and the names of the output arguments (as listed in the function definitions).

The Driver Manager also checks function arguments and state transitions, and for other error conditions before passing the call to the driver associated with the connection. This reduces the amount of error handling that a driver needs to perform. However, the Driver Manager does not check all arguments, state transitions, or error conditions for a given function. For complete information about what the Driver Manager checks, see the following sections, the Diagnostics section of each function in Chapter 22, "ODBC Function Reference," and the state transition tables in Appendix B, "ODBC State Transition Tables."

Validating Arguments

The following general guidelines discuss the arguments or types of arguments checked by the Driver Manager. They are not intended to be exhaustive; the Diagnostics section of each function in Chapter 22, "ODBC Function Reference," lists those SQLSTATEs returned by the Driver Manager for that function. Unless otherwise noted, the Driver Manager returns the return code SQL_ERROR.

- Environment, connection, and statement handles are checked to make sure they are not null pointers and are the correct type of handle for the argument. For example, the Driver Manager checks that the application does not pass an *hdbc* where an *hstmt* is required. If the Driver Manager finds an invalid handle, it returns SQL_INVALID_HANDLE.
- Other required arguments, such as the *phenv* argument in **SQLAllocEnv** or the *szCursor* argument in **SQLSetCursorName**, are checked to make sure they are not null pointers.
- Option flags that cannot be extended by the driver are checked to make sure they specify only supported options. For example, the Driver Manager checks that the *fDriverCompletion* argument in **SQLDriverConnect** is a valid value.
- Option flags that can be extended by the driver, such as the *fInfoType* argument in **SQLGetInfo**, are checked only to make sure that values in the ranges reserved for ODBC options are valid; drivers must check that values in the ranges reserved for driver-specific options are valid. For more information, see "Driver-Specific Data Types, Descriptor Types, Information Types, and Options," later in this chapter.
- All option flags are checked to make sure that no ODBC 2.0 option values are sent to ODBC 1.0 drivers. For example, the Driver Manager returns an error if the *fInfoType* argument in **SQLGetInfo** is SQL_GROUP_BY and the driver is an ODBC 1.0 driver.

- Argument values that specify a column or parameter number are checked to make sure they are greater than 0 or greater than or equal to 0, depending on the function. The driver must check the upper limit of these argument values based on the current result set or SQL statement.
- Buffer length arguments are checked as possible to make sure that their values are appropriate for the corresponding buffer in the context of the given function. For example, *szTableName* in **SQLColumns** is an input argument. Therefore, the Driver Manager checks that if the corresponding length argument (*cbTableName*) is less than 0, it is SQL_NTS. The *szColName* argument in **SQLDescribeCol** is an output argument. Therefore, the Driver Manager checks that the corresponding length argument (*cbColNameMax*) is greater than or equal to 0.

  Note that the driver may also need to check the validity of buffer length arguments. For example, the driver must check that the *cbTableName* argument in **SQLColumns** is less than or equal to the maximum length of a table name in the data source.

Checking State Transitions

The Driver Manager validates the state of the *henv*, *hdbc* or *hstmt* in the context of the function's requirement. For example, an *hdbc* must be in an allocated state before the application can call **SQLConnect** and an *hstmt* must be in a prepared state before the application can call **SQLExecute**.

The state transition tables in Appendix B, "ODBC State Transition Tables," list those state transition errors detected by the Driver Manager for each function. The Driver Manager always returns the SQL_ERROR return code for state transition errors.

Checking for General Errors

The following general guidelines discuss general error checking done by the Driver Manager. They are not intended to be exhaustive; the Diagnostics section of each function in Chapter 22, "ODBC Function Reference," lists those SQLSTATEs returned by the Driver Manager for that function. The Driver Manager always returns the SQL_ERROR return code for general errors.

- Function calls are checked to make sure that the functions are supported by the associated driver.
- The Driver Manager completely implements **SQLDataSources** and **SQLDrivers**. Therefore, it checks for all errors in these functions.
- The Driver Manager checks if a driver implements **SQLGetFunctions**. If the driver does not implement **SQLGetFunctions**, the Driver Manager implements and checks for all errors in it.

- The Driver Manager partially implements **SQLAllocEnv**, **SQLAllocConnect**, **SQLConnect**, **SQLDriverConnect**, **SQLBrowseConnect**, **SQLFreeConnect**, **SQLFreeEnv**, and **SQLError**. Therefore, it checks for some errors in these functions. It may return the same errors as the driver for some of these functions, as both perform similar operations. For example, the Driver Manager or driver may return SQLSTATE IM008 (Dialog failed) if they are unable to display a login dialog box for **SQLDriverConnect**.

Elements of ODBC Functions

The following characteristics apply to all 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). Applications can include variable-length data where appropriate.

C programs that call ODBC functions include the SQL.H, SQLEXT.H, and WINDOWS.H header files. These files define Windows and ODBC constants and types and provide function prototypes for all ODBC functions.

Supporting ODBC Conformance Levels

ODBC defines conformance levels for drivers in two areas: the ODBC API and the ODBC SQL grammar (which includes the ODBC SQL data types). These levels establish standard sets of functionality. By returning the conformance levels it supports, a driver informs applications of the functionality it supports. For a complete discussion of ODBC conformance levels, see "ODBC Conformance Levels" in Chapter 1, "ODBC Theory of Operation."

To claim that it conforms to a given API or SQL conformance level, a driver must support all the functionality in that conformance level, regardless of whether that functionality is supported by the DBMS associated with the driver. A driver may support functionality beyond that in its stated conformance levels.

Note The following sections describe the functions through which a driver returns its conformance levels. Since these are Level 1 extension functions, a given driver may not support them. If a driver does not support these functions, the conformance levels it supports must be included in its documentation.

Supporting API Conformance Levels

ODBC functions are divided into core functions, which are defined in the X/Open and SQL Access Group Call Level Interface specification, and two levels of extension functions, with which ODBC extends this specification. A driver returns its function conformance level through **SQLGetInfo** with the SQL_ODBC_SAG_CLI_CONFORMANCE and SQL_ODBC_API_CONFORMANCE flags. Note that a driver can support one or more extension functions but not conform to ODBC extension Level 1 or 2. The Driver Manager or driver determines and returns whether the driver supports a particular function through **SQLGetFunctions**.

Important Many ODBC applications require that drivers support all of the functions in the Level 1 API conformance level. To ensure that their driver works with most ODBC applications, driver developers should implement all Level 1 functions.

Supporting SQL Conformance Levels

The ODBC SQL grammar, which includes SQL data types, is divided into a minimum grammar, a core grammar, which corresponds to the X/Open and SQL Access Group SQL CAE specification (1992), and an extended grammar, which provides common SQL extensions. A driver returns its SQL conformance level through **SQLGetInfo** with the SQL_ODBC_SQL_CONFORMANCE flag. It returns whether it supports a specific SQL extension through **SQLGetInfo** with a flag for that extension. It returns whether it supports specific SQL data types through **SQLGetTypeInfo**. For more information, see Appendix C, "SQL Grammar," and Appendix D, "Data Types."

Note If a driver supports SQL data types that map to the ODBC SQL date, time, or timestamp data types, the driver must also support the extended SQL grammar for specifying date, time, or timestamp literals.

Buffers

An application passes data to a driver in an input buffer. The driver returns data to an application in an output buffer. The application must allocate memory for both input and output buffers. (If the application will use the buffer to retrieve string data, the buffer must contain space for the null termination byte.)

Caution ODBC does not require drivers to correctly manage buffers that cross segment boundaries in Windows 3.1. The Driver Manager supports the use of such buffers, since it passes buffer addresses to drivers and does not operate on buffer contents. If a driver supports buffers that cross segment boundaries, the documentation for the driver should clearly state this.

If a driver does not support the use of buffers that cross segment boundaries, an application can still use such buffers. The application uses these buffers by passing them to ODBC functions in pieces, none of which crosses a segment boundary.

Input Buffers

An application passes the address and length of an input buffer to a driver. The length of the buffer must be one of the following values:

- A length greater than or equal to zero. This is the actual length of the data in the input buffer. For character data, a length of zero indicates that the data is an empty (zero length) string. Note that this is different from a null pointer. If the application specifies the length of character data, the character data does not need to be null-terminated.
- SQL_NTS. This specifies that a character data value is null-terminated.
- SQL_NULL_DATA. This tells the driver to ignore the value in the input buffer and use a NULL data value instead. It is only valid when the input buffer is used to provide the value of a parameter in an SQL statement.

The operation of ODBC functions on character data containing embedded null characters is undefined, and is not recommended for maximum interoperability. Unless it is specifically prohibited in the description of a given function, the address of an input buffer may be a null pointer. When the address of an input buffer is a null pointer, the value of the corresponding buffer length argument is ignored.

For more information about input buffers, see "Converting Data from C to SQL Data Types" in Appendix D, "Data Types."

Output Buffers

An application passes the following arguments to a driver, so that it can return data in an output buffer:

- The address of the buffer in which the driver returns the data (the output buffer). Unless it is specifically prohibited in a function description, the address of an output buffer can be a null pointer. In this case, the driver does not return anything in the buffer and, in the absence of other errors, returns SQL_SUCCESS.

If necessary, the driver converts data before returning it. The driver always null-terminates character data before returning it.

- The length of the buffer. This is ignored by the driver if the returned data has a fixed length in C, such as an integer, real number, or date structure.
- The address of a variable in which the driver returns the length of the data (the length buffer). The returned length of the data is SQL_NULL_DATA if the data is a NULL value in a result set. Otherwise, it is the number of bytes of data available to return. If the driver converts the data, it is the number of bytes after the conversion. For character data, it does not include the null-termination byte added by the driver.

If the output buffer is too small, the driver attempts to truncate the data. If the truncation does not cause a loss of significant data, the driver returns the truncated data in the output buffer, returns the length of the available data (as opposed to the length of the truncated data) in the length buffer, and returns SQL_SUCCESS_WITH_INFO. If the truncation causes a loss of significant data, the driver leaves the output and length buffers untouched and returns SQL_ERROR. The application calls **SQLError** to retrieve information about the truncation or the error.

For more information about output buffers, see "Converting Data from SQL to C Data Types" in Appendix D, "Data Types."

Environment, Connection, and Statement Handles

When so requested by an application, the Driver Manager and each driver allocate storage for information about the ODBC environment, each connection, and each SQL statement. The handles to these storage areas are returned to the application. The application then uses one or more of them in each call to an ODBC function.

The ODBC interface defines three types of handles:

- The **environment handle** identifies memory storage for global information, including the valid connection handles and current active connection handle. ODBC defines the environment handle as a variable of type HENV. An application uses a single environment handle; it must request this 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 connecting to a a data source. Each connection handle is associated with the environment handle. The environment handle can, however, have multiple connection handles associated with it.
- **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, see Chapter 13, "Establishing Connections." For more information about statement handles, see Chapter 14, "Processing an SQL Statement."

Data Type Support

ODBC defines SQL data types and C data types; a data source may define additional SQL data types. A driver supports these data types in the following ways:

- Accepts SQL and ODBC C data types as arguments in function calls.
- Translates ODBC SQL data types to SQL data types acceptable by the data source, if necessary.
- Converts C data from an application to the SQL data type required by the data source.
- Converts SQL data from a data source to the C data type requested by the application.
- Provides access to data type information through the **SQLDescribeCol** and **SQLColAttributes** functions. If a driver supports them, it also provides data type information through the **SQLGetTypeInfo** and **SQLDescribeParam** functions.

For more information on data types, see Appendix D, "Data Types." The C data types are defined in SQL.H and SQLEXT.H.

ODBC Function Return Codes

When an application calls an ODBC function, the driver executes the function and returns a predefined code. These return codes indicate success, warning, or failure status. The return codes are:

SQL_SUCCESS	SQL_INVALID_HANDLE
SQL_SUCCESS_WITH_INFO	SQL_STILL_EXECUTING
SQL_NO_DATA_FOUND	SQL_NEED_DATA
SQL_ERROR	

If the function returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, the application can call **SQLError** to retrieve additional information. For a complete description of return codes and error handling, see Chapter 16, "Returning Status and Error Information."

Driver-Specific Data Types, Descriptor Types, Information Types, and Options

Drivers can allocate driver-specific values for the following items:

- SQL data types. These are used in the *fSqlType* argument in **SQLBindParameter** and **SQLGetTypeInfo** and returned by **SQLColAttributes**, **SQLColumns**, **SQLDescribeCol**, **SQLGetTypeInfo**, **SQLDescribeParam**, **SQLProcedureColumns**, and **SQLSpecialColumns**.
- Descriptor types. These are used in the *fDescType* argument in **SQLColAttributes**.
- Information types. These are used in the *fInfoType* argument in **SQLGetInfo**.
- Connection and statement options. These are used in the *fOption* argument in **SQLGetConnectOption**, **SQLGetStmtOption**, **SQLSetConnectOption**, and **SQLSetStmtOption**.

For each of these items, there are two ranges of values: a range reserved for use by ODBC, and a range reserved for use by drivers. If you want to implement driver-specific values, such as driver-specific SQL data types or driver-specific statement options, you must reserve a block of values in the driver-specific range. To do this, post a request to the section lead of the ODBC section of the WINEXT forum on CompuServe® or send a request by electronic mail to odbcwish@microsoft.com. Furthermore, you must describe all driver-specific data types, descriptor types, information types, statement options, and connection options in your driver's documentation.

When any of these values is passed to an ODBC function, the Driver Manager checks that values in the ODBC ranges are valid. Drivers must check that values in the driver-specific range are valid. In particular, drivers return SQLSTATE S1C00 (Driver not capable) for driver-specific values that apply to other drivers. The following table shows the ranges of the driver-specific values for each item:

Item	Driver-Specific Range
SQL data types	Less than or equal to SQL_TYPE_DRIVER_START
Descriptor types	Greater than or equal to SQL_COLUMN_DRIVER_START
Information types	Greater than or equal to SQL_INFO_DRIVER_START
Connection and statement options	Greater than or equal to SQL_CONNECT_OPT_DRVR_START

Yielding Control to Windows

Generally, drivers should not explicitly yield control back to Windows. In particular, drivers should not call **PeekMessage** in the Windows API with the PM_REMOVE value set, even when an ODBC function takes a long time, such as when it generates a large result set. Furthermore, driver developers should be careful not to use other, lower-level DLLs (such as network DLLs) that call **PeekMessage**.

If a driver attempted to yield with **PeekMessage**, it would not set the PM_NOYIELD flag. Furthermore, it would not set the PM_NOREMOVE flag, since the first message in the queue is usually for the current application and no other application could be scheduled until that message is removed. However, calling **PeekMessage** with the PM_REMOVE flag causes two problems: the application can be reentered and the **PeekMessage/DispatchMessage** loop in the driver will bypass any preprocessing of messages that may have otherwise been done by the application.

The application can be reentered when the driver dispatches the message; the driver is obligated to dispatch the message because it removed it from the message queue. Because many Windows applications do not guard against reentrancy, this is likely to cause unpredictable results in the application.

Many Windows applications extensively preprocess messages. Because the driver has no knowledge of this preprocessing, it simply calls **DispatchMessage** and is therefore likely to cause unpredictable results in the application. For example, suppose the accelerator key for the Save command is CTRL+S. The message-processing loop in the application calls **TranslateAccelerator** to translate the WM_KEYDOWN and WM_KEYUP messages for CTRL+S into a WM_COMMAND message and to dispatch this message to the menu window. By only calling **DispatchMessage**, the driver does not translate the messages and sends the wrong messages to the menu window.

CHAPTER 18

Constructing an ODBC Driver

This chapter contains a summary of development, debugging, installation, and administration tools provided by the ODBC SDK 2.0.

Testing and Debugging a Driver

The ODBC SDK 2.0 provides the following tools for driver development:

- ODBC Test, an interactive utility that enables you to perform ad hoc and automated testing on drivers. A sample test DLL (the Quick Test) is included which covers basic areas of ODBC driver conformance.
- ODBC Spy, a debugging tool with which you can capture data source information, emulate drivers, and emulate applications.
- A sample driver template written in the C language that illustrates how to write an ODBC driver (16- and 32-bit versions).
- A **#define**, ODBCVER, to specify which version of ODBC you want to compile your driver with. By default, the SQL.H and SQLEXT.H files include all ODBC 2.0 constants and prototypes. To use only the ODBC 1.0 constants and prototypes, add the following line to your driver code before including SQL.H and SQLEXT.H:

  ```
  #define ODBCVER 0x0100
  ```

For additional infomation about the ODBC SDK tools, see the *Microsoft ODBC SDK Guide*.

Installing and Configuring ODBC Software

Users install ODBC software with a setup program and configure the ODBC environment with an administration program. The setup program uses the installer DLL to retrieve information from the ODBC.INF file. This file is created by a driver developer and describes the disks on which the ODBC software is shipped. For more information, see “Constructing the ODBC.INF File” and “Structure of the ODBC.INF File” in Chapter 19, “Installing ODBC Software.”

The administration program uses the installer DLL to configure data sources. The installer DLL calls a *setup DLL* to configure a data source. Driver developers must create a setup DLL for each driver; it may be the driver DLL or a separate DLL. For more information, see Chapter 20, “Configuring Data Sources,” and Chapter 23, “Setup DLL Function Reference.”

CHAPTER 19

Installing ODBC Software

This chapter describes the files that a developer must redistribute in order to enable users to install ODBC software. The ODBC SDK 2.0 provides you with two ways to install ODBC software components (the Driver Manager, drivers, translators, and so on) as follows:

- Use the Driver Setup Toolkit to create a driver setup program, which can be customized by the developer. For more information about using the Driver Setup Toolkit, see the *Microsoft ODBC SDK Guide.*

  Important The Driver Setup Toolkit is a subset of Windows Setup, designed specifically for driver installation. You cannot use it for customizing any other type of installation.

- Create your own setup program. Developers who want to install their own ODBC-enabled applications can use the Windows SDK setup utilities or setup software from other vendors. For more information about the Windows SDK setup utilities, see the Windows SDK documentation.

A setup program makes function calls to the installer DLL. The installer DLL reads information about the ODBC software to be installed from an installation file, ODBC.INF. The installer DLL records information about installed drivers and translators in the ODBCINST.INI file (or registry). The ODBCINST.INI file is used by the Driver Manager to determine which drivers and translators are currently installed. The structure of the ODBC.INF and ODBCINST.INI files is described in the sections that follow.

Redistributing ODBC Files

A number of files are shipped with the ODBC SDK that may be redistributed by application and driver developers. All developers who ship ODBC drivers must redistribute the following files for the specified environments:

ODBC Component	Windows 3.1, WOW (16-bit drivers)	Win32s	WOW (32-bit drivers)	Windows NT
Driver Manager	CTL3DV2.DLL ODBC.DLL	CTL3DV2.DLL ODBC.DLL ODBC16UT.DLL ODBC32.DLL[1]	CTL3D32.DLL ODBC.DLL ODBC16GT.DLL ODBC32GT.DLL	CTL3D32.DLL ODBC32.DLL[1]
Installer	CTL3DV2.DLL ODBC.INF[2] ODBCINST.DLL ODBCINST.HLP	CPN16UT.DLL CTL3DV2.DLL ODBC.INF[2] ODBCCP32.DLL[3] ODBCINST.DLL ODBCINST.HLP	CTL3D32.DLL DS16GT.DLL DS32GT.DLL ODBC.INF[2] ODBCINST.DLL ODBCINST.HLP	CTL3D32.DLL ODBC.INF[2] ODBCCP32.DLL[3] ODBCINST.HLP

1 The ODBC32.DLL file shipped for use with Win32s® is different from the ODBC32.DLL file shipped for use with Windows NT. Under Win32s, it is a thunking layer that calls ODBC16UT.DLL, which in turn calls ODBC.DLL; under Windows NT, it is the Driver Manager. Applications created under Win32s or Windows NT that use the Win32s API will run under either environment.

2 Developers must customize ODBC.INF for the files they ship. For more information, see "Constructing the ODBC.INF File," later in this chapter.

3 The ODBCCP32.DLL file shipped for use with Win32s is different from the ODBCCP32.DLL file shipped for use with Windows NT. Under Win32s, it is a thunking layer that calls CPN16UT.DLL, which in turn calls ODBCINST.DLL; under Windows NT, it is the installer DLL. Applications created under Win32s or Windows NT that use the Win32s API will run under either environment.

Any developers who use the ODBC Driver Setup Toolkit, the program version of the ODBC Administrator, or the ODBC cursor library must redistribute the files required by these components, as listed in the following table. For information about the ODBC Administrator, see the *Microsoft ODBC SDK Guide*. For information about the ODBC cursor library, see Appendix G, "ODBC Cursor Library."

ODBC Component	Windows 3.1, WOW	Windows NT
Driver Setup Toolkit	_BOOTSTP.EXE _MSSETUP.EXE DRVSETUP.EXE [1, 2] SETUP.EXE SETUP.LST [2]	_BOOTSTP.EXE _MSSETUP.EXE DRVSETUP.EXE [1, 2] SETUP.EXE SETUP.LST [2]
Administrator [3]	ODBCADM.EXE	ODBCAD32.EXE
Cursor Library	ODBCCURS.DLL	ODBCCR32.DLL

1 The DRVSETUP.EXE file shipped for setting up ODBC components on Windows 3.1 and WOW is a 16-bit program. The DRVSETUP.EXE file shipped for setting up ODBC components on Windows NT is a 32-bit program. All other files used by the driver setup program are the same.

2 Developers must customize the DRVSETUP.EXE and SETUP.LST files for their product. For more information, see the *Microsoft ODBC SDK Guide*.

3 The ODBC Administrator can be run as a control panel device or as a program on Windows 3.1 or later and on Windows NT. It can be run only as a program on Windows on Windows (WOW).

Creating Your Own Setup Program

If you decide to create your own setup program, it must use the installer DLL shipped with the ODBC SDK. The installer DLL provides functions that a program can call to set up drivers and other ODBC components. The installer can be used to install ODBC components interactively (with a dialog box interface) or silently. For more information about installer DLL functions, see Chapter 24, "Installer DLL Function Reference."

Installing the Software Interactively

To display a dialog box from which a user selects the ODBC components to install (Driver Manager, drivers, and translators), a program calls **SQLInstallODBC** in the installer DLL. It passes a window handle and the full path of the ODBC.INF file to the function. After the user has selected the components to install, **SQLInstallODBC** installs the selected components and records the installed drivers and translators in the ODBCINST.INI file (or registry).

Installing the Software Silently

To silently install the ODBC software, a program calls **SQLInstallODBC** in the installer DLL and passes it a null window handle, the full path of the ODBC.INF file, and, optionally, a list of drivers to install. **SQLInstallODBC** installs the drivers in the list (if any), the Driver Manager, and any translators that are listed in the ODBC.INF file and records the installed drivers and translators in the ODBCINST.INI file (or registry).

Installing Individual ODBC Components

A program can also install individual ODBC components. To install the Driver Manager, a program first calls **SQLInstallDriverManager** in the installer DLL to get the target directory for the Driver Manager. This is usually the directory in which Windows DLLs reside. The program then uses the information in the [ODBC Driver Manager] section of the ODBC.INF file to copy the Driver Manager and related files from the installation disk to this directory.

To install an individual driver, a program first calls **SQLInstallDriver** in the installer DLL to add the driver specification to the ODBCINST.INI file. **SQLInstallDriver** returns the driver's target directory—usually the directory in which Windows DLLs reside. The program then uses the information in the driver's section of the ODBC.INF file to copy the driver DLL and related files from the installation disk to this directory.

Constructing the ODBC.INF File

The ODBC.INF file must be constructed by anyone who ships drivers. The ODBC.INF file shipped with the ODBC SDK may be used as a template. Particular care should be taken to ensure that:

- All disks are described in the [Source Media Descriptions] section and each entry is placed in double quotation marks (") and separated by commas.
- The [ODBC Driver Manager] and [ODBC] sections are not modified and the [ODBC Administrator] section is not modified if the ODBC Administrator is being used.
- All drivers are listed in the [ODBC Drivers] section, each driver has a section describing all the files it needs, and each entry in the driver specification section has the correct number of commas.
- There is a data source specification section for each data source listed in a driver keyword section.
- All translators are listed in the [ODBC Translators] section, each translator has a section describing all the files it needs, and each entry in the translator specification section has the correct number of commas.
- The ODBC.INF file does not contain any tab characters.

Structure of the ODBC.INF File

The ODBC.INF file contains the following sections:

- The [Source Media Descriptions] section describes the disks used to install the ODBC software.
- The [ODBC Driver Manager] section describes the files shipped for the Driver Manager.
- The [ODBC] section describes the files shipped for the installer DLL.
- The [ODBC Administrator] section describes the files shipped for the ODBC Administrator.
- The [ODBC Drivers] and [ODBC Translators] sections describes the ODBC drivers and translators shipped on the disk.
- For each driver described in the [ODBC Drivers] section, there is a section that describes the files shipped for that driver and an optional section that lists driver attribute keywords. For each data source listed in a driver keyword section, there is a section that describes the data source. For each translator described in the [ODBC Translator] section, there is a section that describes the files shipped for that translator.

[Source Media Descriptions] Section

The entries in the [Source Media Descriptions] section describe each of the shipped disks. Each must be enclosed in double quotation marks (") and separated by a comma (,). The format of the section is:

[Source Media Descriptions]
"*disk-ID-number1*","*disk-label1*","*tag-filename1*","*setup.exe-rel-path*"
"*disk-ID-number2*","*disk-label2*","*tag-filename2*","*setup.exe-rel-path*"

.
.
.

where each of the arguments has the following meaning:

Argument	Meaning
disk-ID-number	Disk identification number. A unique integer from 1 to 999.
disk-label	Disk label.
tag-filename	The name of a file residing on the disk. The setup program uses this to check that the correct disk has been placed in the drive.
setup.exe-rel-path	The relative path of SETUP.EXE. (This is used only if the installable files reside on a network disk drive.)

For example, if the ODBC software is shipped on a single disk, this section might be:

```
[Source Media Descriptions]
"1","ODBC Setup","SETUP.EXE","."
```

[ODBC Drivers] Section

The [ODBC Drivers] section lists the descriptions of the shipped drivers. A driver description is usually the name of the DBMS associated with that driver. Each entry in the section must start in column 1. The format of the section is:

[ODBC Drivers]
"*driver-desc1*"=
"*driver-desc2*"=

.
.
.

For example, suppose drivers for formatted text files and SQL Server are shipped. The [ODBC Drivers] section might contain the following entries:

```
[ODBC Drivers]
"Text"=
"SQL Server"=
```

Driver Specification Sections

For each driver in the [ODBC Drivers] section, a separate section lists the disk location, name, and installation properties of each file needed by the driver. The section name is the driver description as listed in the [ODBC Drivers] section. The driver DLL must be listed with the **Driver** keyword. If there is a separate setup DLL, it must be listed with the **Setup** keyword. All other files, such as network communication DLLs and driver data files, must be listed with their own keywords. If any of these other files are to be placed in the \WINDOWS (as opposed to the \WINDOWS\SYSTEM) directory, they must use the **Windows*nn*** keyword, where *nn* is a number from 00 to 99. The format of a driver specification section is:

[*driver-desc*]
"Driver"=*driver-disk-ID,driver-DLL-filename,,,,Date,,,Replace,,,,,,*
➥ *Shared,Size,,,,Version,*
[**"Setup"**=*setup-disk-ID,setup-DLL-filename,,,,Date,,,Replace,,,,,,*
➥ *Shared,Size,,,,Version,*]
["*keyword3*"=*disk-ID3,filename3,,,,Date,,,Replace,,,,,,Shared,Size,,,,Version,*]
["*keyword4*"=*disk-ID4,filename4,,,,Date,,,Replace,,,,,,Shared,Size,,,,Version,*]

•

•

•

where *Date*, *Replace*, *Shared*, *Size*, and *Version* are installation properties and non-bold brackets ([]) indicate optional keywords. All commas (20 per line) must be included, even if the properties are left blank. For more information, see "Installation Properties" later in this section.

For example, suppose that a driver for formatted text files is created on May 11, 1992, has a separate setup DLL, and is shipped on the first installation disk. Suppose also that a driver for SQL Server is created on May 15, 1992, does not have a separate setup DLL, requires DBNMP3.DLL (a Microsoft SQL Server Net-Library file) for network communications, and is shipped on the second installation disk. The driver specification sections for these drivers might be:

```
[Text]
"Driver"=1,TEXT.DLL,,,,1992-05-11,,,,,,,,,,89302,,,,01.00.17.11,
"Setup"=1,TXTSETUP.DLL,,,,1992-05-11,,,,,,,,,,9601,,,,01.00.17.11,

[SQL Server]
"Driver"=2,SQLSRVR.DLL,,,,1992-05-15,,,,,,,,,,95264,,,,01.00.17.15,
"Sqlnet"=2,DBNMP3.DLL,,,,1992-05-15,,,,,,,,,SHARED,7473,,,,01.00.17.15,
```

Driver Keyword Sections

For each driver in the [ODBC Drivers] section, a separate section lists the driver attribute keywords. If a driver does not have any keywords that describe it, this section should not be included in the ODBC.INF file.

For each data source listed with the **CreateDSN** keyword, **SQLInstallODBC** creates a data source in the ODBC.INI file (or registry). For all other driver keywords, **SQLInstallODBC** copies the keywords and their values to the driver's specification section in the ODBCINST.INI file (or registry). To find out information about a driver before connecting to it, an application retrieves these keywords by calling **SQLDrivers**.

The format of a driver keyword section is:

[*driver-desc*-**Keys**]
[**APILevel=0 | 1 | 2**]
[**CreateDSN=***data-source-name*[,*data-source-name*]...]
[**ConnectFunctions=**{**Y**|**N**}{**Y**|**N**}{**Y**|**N**}]
[**DriverODBCVer=01.00 | 02.00**]
[**FileUsage=0 | 1 | 2**]
[**FileExtns=*.***file-extension1*[,***.***file-extension2*]...]
[**SQLLevel=0 | 1 | 2**]

where the use of each keyword is:

Keyword	Usage
APILevel	A number indicating the ODBC API conformance level supported by the driver: 0 = None 1 = Level 1 supported 2 = Level 2 supported This must be the same as the value returned for the SQL_ODBC_API_CONFORMANCE information type in **SQLGetInfo**.
CreateDSN	The name of one or more data sources to be created when the driver is installed. The ODBC.INF file must include one data source specification section for each data source listed with the **CreateDSN** keyword. These sections should not include the **Driver** keyword, since this is specified in the driver specification section, but must include enough information for the **ConfigDSN** function in the driver-specific setup DLL to create a data source specification without displaying any dialog boxes. For the format of a data source specification section, see "Data Source Specification Sections" in Chapter 20, "Configuring Data Sources."

Keyword	Usage
ConnectFunctions	A three-character string indicating whether the driver supports **SQLConnect**, **SQLDriverConnect**, and **SQLBrowseConnect**. If the driver supports **SQLConnect**, the first character is "Y"; otherwise, it is "N". If the driver supports **SQLDriverConnect**, the second character is "Y"; otherwise, it is "N". If the driver supports **SQLBrowseConnect**, the third character is "Y"; otherwise, it is "N". For example, if a driver supports **SQLConnect** and **SQLDriverConnect**, but not **SQLBrowseConnect**, this is "YYN".
DriverODBCVer	A character string with the version of ODBC that the driver supports. The version is of the form ##.##, where the first two digits are the major version and the next two digits are the minor version. For the version of ODBC described in this manual, the driver must return "02.00". This must be the same as the value returned for the SQL_DRIVER_ODBC_VER information type in **SQLGetInfo.**
FileUsage	A number indicating how a single-tier driver directly treats files in a data source. 0 = The driver is not a single-tier driver. For example, an ORACLE driver is a two-tier driver. 1 = A single-tier driver treats files in a data source as tables. For example, an Xbase driver treats each Xbase file as a table. 2 = A single-tier driver treats files in a data source as a qualifier. For example, a Microsoft Access® driver treats each Microsoft Access file as a complete database. An application might use this to determine how users will select data. For example, Xbase and Paradox® users often think of data as stored in files, while ORACLE and Microsoft Access users generally think of data as stored in tables. When a user selects Open Data File from the File menu, an application could display the Windows File Open common dialog box. The list of file types would use the file extensions specified with the **FileExtns** keyword for drivers that specify a **FileUsage** value of 1 and "Y" as the second character of the value of the **ConnectFunctions** keyword. After the user selects a file, the application would call **SQLDriverConnect** with the **DRIVER** keyword, then execute a **SELECT * FROM** ***table-name*** statement. When the user selects Import Data from the File menu, an application could display a list of descriptions for drivers that specify a **FileUsage** value of 0 or 2 and "Y" as the second character of the value of the **ConnectFunctions** keyword. After the user selects a driver, the application would call **SQLDriverConnect** with the **DRIVER** keyword, then display a custom Select Table dialog box.

Keyword	Usage
FileExtns	For single-tier drivers, a comma-separated list of extensions of the files the driver can use. For example, a dBASE driver might specify *.dbf and a formatted text file driver might specify *.txt,*.csv. For an example of how an application might use this information, see the **FileUsage** keyword.
SQLLevel	A number indicating the ODBC SQL conformance level supported by the driver: 0 = Minimum grammar 1 = Core grammar 2 = Extended grammar This must be the same as the value returned for the SQL_ODBC_SQL_CONFORMANCE information type in **SQLGetInfo.**

For example, suppose a driver for formatted text files can use files with the .TXT and .CSV extensions and that a data source for this driver is to be created when it is installed. The driver keyword and data source specification sections might be:

```
[Text-Keys]
CreateDSN=Text Files
FileExtns=*.txt,*.csv
FileUsage=1

[Text Files]
Directory=#current directory#
TextFormat=Comma Delimited
```

[ODBC Translators] Section

The [ODBC Translators] section lists the descriptions of the shipped translators. Each entry in the section must start in column 1. The format of the section is:

[ODBC Translators]
"*translator-desc1*"=
"*translator-desc2*"=
•
•
•

For example, suppose only the Microsoft Code Page Translator is shipped. The [ODBC Translators] section might contain the following entry:

```
[ODBC Translators]
"MS Code Page Translator"=
```

Translator Specification Sections

For each translator in the [ODBC Translator] section, a separate section lists the disk location, name, and installation properties of each file needed by the translator. The section name is the translator description as listed in the [ODBC Translators] section. The translation DLL must be listed with the **Translator** keyword. If there is a separate translator setup DLL, it must be listed with the **Setup** keyword. All other files, such as translation tables for a code page translator, must be listed with their own keywords. If any of these other files are to be placed in the \WINDOWS directory, they must use the **Windows*nn*** keyword, where *nn* is a number from 00 to 99. The format of a translator specification section is:

[*translator-desc*]
"Translator"=*translator-disk-ID,translator-DLL-filename,,,,Date,,,Replace,,,,,,*
➥ *Shared,Size,,,,Version,*
[**"Setup"**=*setup-disk-ID,setup-DLL-filename,,,,Date,,,Replace,,,,,,*
➥ *Shared,Size,,,,Version,*]
["*keyword3*"=*disk-ID3,filename3,,,,Date,,,Replace,,,,,,Shared,Size,,,,Version,*]
["*keyword4*"=*disk-ID4,filename4,,,,Date,,,Replace,,,,,,Shared,Size,,,,Version,*]

•
•
•

where *Date*, *Replace*, *Shared*, *Size*, and *Version* are installation properties and non-bold brackets ([]) indicate optional keywords. All commas (20 per line) must be included, even if the properties are left blank. For more information, see "Installation Properties" in the following section.

For example, suppose the Microsoft Code Page Translator is shipped with translation tables for the Multilingual (850) and Nordic (865) code pages. Suppose also that it does not have a separate setup DLL, that it requires the CTL3D.DLL file, and that the files are created on June 1, 1993. The translator specification section for this translator might be:

```
[MS Code Page Translator]
"Translator"=1,MSCPXLT.DLL,,,,1993-06-01,,,,,,,,,,10512,,,,01.01.27.25,
"Ctl3d"=1,CTL3D.DLL,,,,1993-06-01,,,,,,,,,SHARED,14480,,,,1.1.3.0,
"Code Page 850"=1,10070850.CPX,,,,1993-06-01,,,,,,,,,,2216,,,,,
"Code Page 865"=1,10070865.CPX,,,,1993-06-01,,,,,,,,,,2130,,,,,
```

Installation Properties

The following table describes the installation properties used in the driver specification and translator specification sections. A *blank* value means no value was specified for the property.

Property	Possible values	Meaning
Date	*blank*	The file has no date and is treated as if it was created before all other files.
	date in the format YYYY-MM-DD	Date the file was created. This should match the date in the version.
Replace	ALWAYS	Always overwrite an existing copy of the file.
	NEVER	Never overwrite an existing copy of the file.
	blank or OLDER	Overwrite an existing copy of the file if it is older.
	UNPROTECTED	Overwrite an existing copy of the file if it is unprotected.
Shared	*blank*	Do not treat the file as if it is a shared file.
	SHARED	Treat the file as if it is a shared file, that is, as if non-ODBC programs use it.
Size	*integer*	Approximate size of the uncompressed file in bytes.
	blank	0
Version	*blank*	00.00.00.00
	VV.vv.mm.dd	Version number of the file. The format is: VV: major version number vv: minor version number mm: month file was created (may be greater than 12) dd: date file was created

Structure of the ODBCINST.INI File

The ODBCINST.INI file is a Windows initialization file used in Windows 3.1 and WOW that contains the following sections:

- The [ODBC Drivers] section lists the description of each available driver.
- For each driver described in the [ODBC Drivers] section, there is a section that lists the driver DLL, the setup DLL, and any driver attribute keywords.
- An optional section that specifies the default driver.

- The [ODBC Translators] section lists the description of each available translator.
- For each translator described in the [ODBC Translators] section, there is a section that lists the translator DLL and the setup DLL.

On Windows NT, this information is stored in the registry. The key structure in which it is stored is:

```
HKEY_LOCAL_MACHINE
   Software
      ODBC
         ODBCINST.INI
```

A subkey of the ODBCINST.INI subkey is created for each section of the ODBCINST.INI file. A value is added to this subkey for each keyword-value pair in the section. The value's name is the same as the keyword, the value's data is the same as the value associated with the keyword, and the value's type is REG_SZ.

Note This section uses terminology for Windows initialization files. For the registry, you should substitute *ODBCINST.INI subkey* for *ODBCINST.INI file*, *subkey* for *section*, *value* for *keyword-value pair*, *value name* for *keyword*, and *value data* for *value*.

For information on the general structure of Windows initialization files, see the Windows SDK documentation. For information on the Windows NT registry, see the Windows NT SDK documentation.

[ODBC Drivers] Section

The [ODBC Drivers] section lists the descriptions of the installed drivers. A driver description is usually the name of the DBMS associated with that driver. Each entry in the section also states that the driver is installed (no other options are allowed). The format of the section is:

[ODBC Drivers]
driver-desc1=**Installed**
driver-desc2=**Installed**
.
.
.

For example, suppose a user has installed drivers for formatted test files and SQL Server. The [ODBC Drivers] section might contain the following entries:

```
[ODBC Drivers]
Text=Installed
SQL Server=Installed
```

Driver Specification Sections

Each driver described in the [ODBC Drivers] section has a section of its own. The section name is the driver description from the [ODBC Drivers] section. It lists the full paths of the driver and setup DLLs, which are the same if the setup function is in the driver DLL. It also lists any driver attribute keywords. The format of a driver specification section is:

[*driver-desc*]
Driver=*driver-DLL-path*
Setup=*setup-DLL-path*
[**APILevel=0 | 1 | 2**]
[**ConnectFunctions=**{**Y**|**N**}{**Y**|**N**}{**Y**|**N**}]
[**DriverODBCVer=01.00 | 02.00**]
[**FileUsage=0 | 1 | 2**]
[**FileExtns=*.***file-extension1*[**,*.***file-extension2*]...]
[**SQLLevel=0 | 1 | 2**]

For information about driver attribute keywords, see "Driver Keyword Sections" earlier in this chapter.

For example, suppose driver for formatted text files has a driver DLL named TEXT.DLL and a setup DLL named TXTSETUP.DLL, and that it can use files with the .TXT and .CSV extensions. Suppose also that a SQL Server driver has a driver DLL named SQLSRVR.DLL, which contains the setup function. The specification sections for these drivers might be:

```
[Text]
Driver=C:\WINDOWS\SYSTEM\TEXT.DLL
Setup=C:\WINDOWS\SYSTEM\TXTSETUP.DLL
FileExtns=*.txt,*.csv
FileUsage=1

[SQL Server]
Driver=C:\WINDOWS\SYSTEM\SQLSRVR.DLL
Setup=C:\WINDOWS\SYSTEM\SQLSRVR.DLL
```

Because the driver and setup DLLs are different for the formatted text file driver, two different files are listed; because they are the same for the SQL Server driver, the same file is listed twice.

Default Driver Specification Section

The ODBCINST.INI file may contain a default driver specification section. The section must be named [Default]. It contains a single entry, which gives the description of the default driver, which is the driver used by the default data source. (This driver must also be described in the [ODBC Drivers] section and in a driver specification section of its own.) The format of the default driver specification section is:

[Default]
Driver=*default-driver-desc*

For example, if the SQL Server driver is the default driver, the default driver specification section might be:

```
[Default]
Driver=SQL Server
```

[ODBC Translators] Section

The [ODBC Translators] section lists the descriptions of the installed translators. Each entry in the section also states that the translator is installed (no other options are allowed). The format of the section is:

[ODBC Translators]
*translator-desc1***=Installed**
*translator-desc2***=Installed**

•
•
•

For example, suppose a user has installed the Microsoft Code Page Translator and a custom ASCII to EBCDIC translator. The [ODBC Translators] section might contain the following entries:

```
[ODBC Translators]
MS Code Page Translator=Installed
ASCII to EBCDIC=Installed
```

Translator Specification Sections

Each translator described in the [ODBC Translators] section has a section of its own. The section name is the translator description from the [ODBC Translator] section. It lists the full paths of the translation and setup DLLs, which are the same if the setup function is in the translator DLL. The format of a translator specification section is:

[*translator-desc*]
Driver=*translator-DLL-path*
Setup=*setup-DLL-path*

For example, suppose the Microsoft Code Page Translator has a translation DLL named MSCPXLT.DLL, which contains the setup function. Suppose also that a custom ASCII to EBCDIC translator has a translation DLL named ASCEBC.DLL and a setup DLL named ASCEBCST.DLL. The specification sections for these translators might be:

```
[MS Code Page Translator]
Translator=C:\WINDOWS\SYSTEM\MSCPXLT.DLL
Setup=C:\WINDOWS\SYSTEM\MSCPXLT.DLL

[ASCII to EBCDIC]
Translator=C:\WINDOWS\SYSTEM\ASCEBC.DLL
Setup=C:\WINDOWS\SYSTEM\ASCEBCST.DLL
```

Because the translator and setup DLLs are the same for the Microsoft Code Page Translator, the same file is listed twice; because they are different for the ASCII to EBCDIC translator, two different files are listed.

CHAPTER 20

Configuring Data Sources

This chapter describes the files that a developer must redistribute in order to enable users to configure ODBC data sources. The ODBC SDK 2.0 provides you with two ways to configure ODBC data sources, as follows:

- Use the ODBC Administrator (available as a program or as a Control Panel item). For more information about using the ODBC Administrator, see the *Microsoft ODBC SDK Guide.*
- Create your own program to configure data sources.

A program that configures data sources makes function calls to the installer DLL. The installer DLL calls a setup DLL to configure a data source. There is one setup DLL for each driver; it may be the driver DLL or a separate DLL. The setup DLL prompts the user for information that the driver needs to connect to the data source and the default translator, if supported. It then calls the installer DLL and the Windows API to record this information in the ODBC.INI file (or registry). The structure of the ODBC.INI file is described in the sections that follow.

Creating Your Own Data Source–Management Program

If you decide to create your own data source–management program, it must use the installer DLL shipped with the ODBC SDK. The installer DLL provides functions that a program can call to add, modify, and delete data sources, remove a default data source, and select a translator. These functions work with the driver's setup DLL, which returns the information a driver needs to connect to a data source, and the translator's setup DLL, which returns a default translation option for a data source. For more information about installer DLL functions, see Chapter 24, "Installer DLL Function Reference."

Adding, Modifying, and Deleting Data Sources

To display a dialog box with which a user can add, modify, and delete data sources, a program calls **SQLManageDataSources** in the installer DLL. This is the function that is invoked when the installer DLL is called from the Control Panel. To add, modify, or delete a data source, **SQLManageDataSources** calls **ConfigDSN** in the setup DLL for the driver associated with that data source.

To directly add, modify, or delete data sources, a program calls **SQLConfigDataSource** in the installer DLL. The program passes the name of the data source and an option that specifies the action to take. **SQLConfigDataSource** calls **ConfigDSN** in the setup DLL and passes it the arguments from **SQLConfigDataSource**.

For more information, see Chapter 23, "Setup DLL Function Reference," and Chapter 24, "Installer DLL Function Reference."

Specifying a Default Data Source

The default data source is the same as any other data source, except that it has the name Default. (Hence, the connection information includes the keyword-value pair DSN=Default.) To add or modify a default data source, a program performs the same steps that it does to add or modify any other data source. To remove the default data source, a program calls **SQLRemoveDefaultDataSource** in the installer DLL.

Specifying a Default Translator

If a driver supports translators, its setup DLL must provide a way for users to select the default translator and default translation option for a data source. To do this, a setup DLL can call **SQLGetTranslator**. This function displays a list of all installed translators. If the user selects a translator, **SQLGetTranslator** calls **ConfigTranslator** in the selected translator's setup DLL to get the default translation option. The user can also specify that there is no default translator. **SQLGetTranslator** returns the name, path, and option of the default translator (if any).

To add, modify, or delete the default translator and default translation option specified in the ODBC.INI file (or registry), the driver's setup DLL calls **SQLWritePrivateProfileString** in the installer DLL for the **TranslationName**, **TranslationDLL**, and **TranslationOption** keywords. These keywords have the following values:

Keyword	Value
TranslationName	Name of the translator as listed in the [ODBC Translators] section of the ODBCINST.INI file (or registry).
TranslationDLL	Full path of the translation DLL.
TranslationOption	ASCII representation of the 32-bit integer translation option.

Structure of the ODBC.INI File

The ODBC.INI file is a Windows initialization file used in Windows 3.1 and WOW. It is created by the installer DLL when data sources are first configured and contains the following sections:

- The [ODBC Data Sources] section lists the name of each available data source and the description of its associated driver.
- For each data source listed in the [ODBC Data Sources] section, there is a section that lists additional information about that data source.
- An optional section that specifies the default data source.
- An optional section that specifies ODBC options.

On Windows NT, this information is stored in the registry. The key structure in which it is stored is:

```
HKEY_CURRENT_USER
   Software
      ODBC
         ODBC.INI
```

A subkey of the ODBC.INI subkey is created for each section of the ODBC.INI file. A value is added to this subkey for each keyword-value pair in the section. The value's name is the same as the keyword, the value's data is the same as the value associated with the keyword, and the value's type is REG_SZ.

Note This section uses terminology for Windows initialization files. For the registry, you should substitute *ODBC.INI subkey* for *ODBC.INI file*, *subkey* for *section*, *value* for *keyword-value pair*, *value name* for *keyword*, and *value data* for *value*.

For information on the general structure of Windows initialization files, see the Windows SDK documentation. For information on the Windows NT registry, see the Windows NT SDK documentation.

[ODBC Data Sources] Section

The [ODBC Data Sources] section lists the data sources specified by the user. Each entry in the section lists a data source and the description of the driver it uses. The driver description is usually the name of the associated DBMS. The format of the section is:

[ODBC Data Sources]
data-source-name1=driver-desc1
data-source-name2=driver-desc2

•
•
•

For example, suppose a user has three data sources: Personnel and Inventory, which use formatted text files, and Payroll, which uses an SQL Server DBMS. The [ODBC Data Sources] section might contain the following entries:

```
[ODBC Data Sources]
Personnel=Text
Inventory=Text
Payroll=SQL Server
```

Data Source Specification Sections

Each data source listed in the [ODBC Data Sources] section has a section of its own. The section name is the data source name from the [ODBC Data Sources] section. It must list the driver DLL and may list a description of the data source. If the driver supports translators, the section may list the name of a default translator, the default translation DLL, and the default translation option. The section may also list other information required by the driver to connect to the data source. For example, the driver might require a server name, database name, or schema name.

The format of a data source specification section is:

[*data-source-name*]
Driver=*driver-DLL-path*
[**Description=***data-source-desc*]
[**TranslationDLL=***translation-DLL-path*]
[**TranslationName=***translator-name*]
[**TranslationOption=***translation-option*]
[*keyword1=string1*]
[*keyword2=string2*]

•
•
•

where brackets ([]) indicate optional keywords.

For example, suppose an Rdb driver requires the ID of the last user to log in, a server name, and a schema declaration statement. Suppose also that the Personnel data source uses the Microsoft Code Page Translator to translate between the Windows Latin 1 (1007) and Multilingual (850) code pages. The data source specification sections for the Personnel and Inventory data sources might be:

```
[Personnel]
Driver=C:\WINDOWS\SYSTEM\RDB.DLL
Description=Personnel database: CURLY
TranslationName=MS Code Page Translator
TranslationDLL=C:\WINDOWS\SYSTEM\MSCPXLT.DLL
TranslationOption=10070850
Lastuid=smithjo
Server=curly
Schema=declare schema personnel filename
➥ "sys$sysdevice:[corpdata]personnel.rdb"

[Inventory]
Driver=C:\WINDOWS\SYSTEM\RDB.DLL
Description=Western Region Inventory
Lastuid=smithjo
Server=larry
Schema=declare schema inventory filename
➥ "sys$sysdevice:[regionw]inventory.rdb"
```

Note that more than one data source can use the same driver.

Default Data Source Specification Section

The ODBC.INI file may contain a default data source specification section. The data source must be named Default and is not listed in the [ODBC Data Sources] section. The format of the default data source specification section is the same as the structure of any other data source specification section.

ODBC Options Section

The ODBC.INI file may contain a section that specifies ODBC options. These options are set in the Options dialog box displayed by the **SQLManageDataSources** function. The format of the ODBC options section is:

[ODBC]
Trace=0 | 1
TraceFile=*tracefile-path*
TraceAutoStop=0 | 1

where each keyword has the following meaning:

Keyword	Meaning
Trace	If the **Trace** keyword is set to 1 when an application calls **SQLAllocEnv**, then tracing is enabled. On Windows and WOW, it is enabled for all ODBC-enabled applications. On Windows NT, it is enabled only for the calling application. If the **Trace** keyword is set to 0 when an application calls **SQLAllocEnv**, then tracing is disabled. On Windows and WOW, it is disabled for all ODBC-enabled for all applications. On Windows NT, it is disabled only for the calling application. This is the default value. An application can enable or disable tracing with the SQL_OPT_TRACE connection option. However, doing so does not change the value of this keyword.
TraceFile	If tracing is enabled, the Driver Manager writes to the trace file specified by the **TraceFile** keyword. If no trace file is specified, the Driver Manager writes to the \SQL.LOG file in the current directory. This is the default value. On Windows NT, tracing should only be used for a single application or each application should specify a different trace file. Otherwise, two or more applications will attempt to open the same trace file at the same time, causing an error. An application can specify a new trace file with the SQL_OPT_TRACEFILE connection option. However, doing so does not change the value of this keyword.
TraceAutoStop	If the **TraceAutoStop** keyword is set to 1 when an application calls **SQLFreeEnv**, then tracing is disabled for all applications and the **Trace** keyword is set to 0. On Windows and WOW, it is disabled for all applications; on Windows NT, it is disabled for the calling application and any applications started after the application calls **SQLFreeEnv**. This is the default value. If the **TraceAutoStop** keyword is set to 0, then tracing must be disabled with the Options dialog box displayed by the **SQLManageDataSources** function.

CHAPTER 21

Function Summary

This chapter summarizes the functions used by ODBC-enabled applications and related software:

- ODBC functions
- Setup DLL functions
- Installer DLL functions
- Translation DLL functions

ODBC Function Summary

The following table lists ODBC functions, grouped by type of task, and includes the conformance designation and a brief description of the purpose of each function. For more information about conformance designations, see "ODBC Conformance Levels" in Chapter 1, "ODBC Theory of Operation." For more information about the syntax and semantics for each function, see Chapter 22, "ODBC Function Reference."

An application can call the **SQLGetInfo** function to obtain conformance information about a driver. To obtain information about support for a specific function in a driver, an application can call **SQLGetFunctions**.

Task	Function Name	Conformance	Purpose
Connecting to a Data Source	**SQLAllocEnv**	Core	Obtains an environment handle. One environment handle is used for one or more connections.
	SQLAllocConnect	Core	Obtains a connection handle.
	SQLConnect	Core	Connects to a specific driver by data source name, user ID, and password.
	SQLDriverConnect	Level 1	Connects to a specific driver by connection string or requests that the Driver Manager and driver display connection dialog boxes for the user.
	SQLBrowseConnect	Level 2	Returns successive levels of connection attributes and valid attribute values. When a value has been specified for each connection attribute, connects to the data source.
Obtaining Information about a Driver and Data Source	**SQLDataSources**	Level 2	Returns the list of available data sources.
	SQLDrivers	Level 2	Returns the list of installed drivers and their attributes.
	SQLGetInfo	Level 1	Returns information about a specific driver and data source.
	SQLGetFunctions	Level 1	Returns supported driver functions.
	SQLGetTypeInfo	Level 1	Returns information about supported data types.
Setting and Retrieving Driver Options	**SQLSetConnectOption**	Level 1	Sets a connection option.
	SQLGetConnectOption	Level 1	Returns the value of a connection option.
	SQLSetStmtOption	Level 1	Sets a statement option.
	SQLGetStmtOption	Level 1	Returns the value of a statement option.

Task	Function Name	Conformance	Purpose
Preparing SQL Requests	**SQLAllocStmt**	Core	Allocates a statement handle.
	SQLPrepare	Core	Prepares an SQL statement for later execution.
	SQLBindParameter	Level 1	Assigns storage for a parameter in an SQL statement.
	SQLParamOptions	Level 2	Specifies the use of multiple values for parameters.
	SQLGetCursorName	Core	Returns the cursor name associated with a statement handle.
	SQLSetCursorName	Core	Specifies a cursor name.
	SQLSetScrollOptions	Level 2	Sets options that control cursor behavior.
Submitting Requests	**SQLExecute**	Core	Executes a prepared statement.
	SQLExecDirect	Core	Executes a statement.
	SQLNativeSql	Level 2	Returns the text of an SQL statement as translated by the driver.
	SQLDescribeParam	Level 2	Returns the description for a specific parameter in a statement.
	SQLNumParams	Level 2	Returns the number of parameters in a statement.
	SQLParamData	Level 1	Used in conjunction with **SQLPutData** to supply parameter data at execution time. (Useful for long data values.)
	SQLPutData	Level 1	Send part or all of a data value for a parameter. (Useful for long data values.)

Task	Function Name	Conformance	Purpose
Retrieving Results and Information about Results	**SQLRowCount**	Core	Returns the number of rows affected by an insert, update, or delete request.
	SQLNumResultCols	Core	Returns the number of columns in the result set.
	SQLDescribeCol	Core	Describes a column in the result set.
	SQLColAttributes	Core	Describes attributes of a column in the result set.
	SQLBindCol	Core	Assigns storage for a result column and specifies the data type.
	SQLFetch	Core	Returns a result row.
	SQLExtendedFetch	Level 2	Returns multiple result rows.
	SQLGetData	Level 1	Returns part or all of one column of one row of a result set. (Useful for long data values.)
	SQLSetPos	Level 2	Positions a cursor within a fetched block of data.
	SQLMoreResults	Level 2	Determines whether there are more result sets available and, if so, initializes processing for the next result set.
	SQLError	Core	Returns additional error or status information.