SOLID™ Programmer Guide

June, 2000
Version 3.51

SOLID

Solid Information Technology Ltd.
www.solidtech.com
sales@solidtech.com;techsupp@solidtech.com

Document number SSPG-3.51-0600
Date: June 26, 2000

Contents

Welcome

SOLID is a data management product for today's smart networks.

SOLID provides support for real-time operating systems such as VxWorks and ChorusOS, and for preferred platforms such as Windows 98/NT, Linux, Solaris, HP-UX and other UNIX platforms. It also provides the features you would expect to find in any industrial-strength database server—multithread architecture, stored procedures, optimistic row level transaction management, but delivered with the special needs of today's applications.

About this Guide

The **SOLID Programmer Guide** contains information about using the different Application Programming Interfaces with SOLID *Embedded Engine*™ or SOLID *SynchroNet*™.

SOLID *ODBC Driver*, SOLID *Light Client* and SOLID *JDBC Driver*, are available for application development purposes. SOLID's 32-bit native *ODBC Driver* conforms to the Microsoft ODBC 3.5.x API standard. SOLID *Light Client* is a lightweight version of the SOLID *ODBC API and i*s intended for environments where the footprint of the client application is critical. The SOLID *JDBC Driver* is a SOLID implementation of the JDBC 2.0 standard.

Organization

This manual contains the following chapters:

- *Chapter 1, Introduction to SOLID APIs,* provides an overview of the application programming interfaces available for accessing SOLID databases.
- *Chapter 2, Using SOLID ODBC API,* provides SOLID-specific information for developing applications with ODBC API.
- *Chapter 3, Stored Procedures, Events, Triggers, and Sequences,* explains advanced features for developing applications using SOLID.

- *Chapter 4, Using UNICODE*, describes how to implement the UNICODE standard, providing the capability to encode characters used in the major languages of the world.
- *Chapter 5, Using SOLID Light Client*, describes how to use SOLID *Light Client*, and API especially designed for implementing embedded solutions with limited memory resources.
- *Chapter 6, Using the SOLID JDBC Driver*, describes how to use the SOLID J*DBC Driver*, a 100% Pure Java™ implementation of the Java Database Connectivity (JDBC™) standard.

The *Appendixes* give you detailed information about error messages, data types, and SOLID SQL functionality, etc.

Audience

This guide assumes a working knowledge of the C and Java programming languages, general DBMS knowledge, and a familiarity with SQL, SOLID *Embedded Engine* or SOLID *SynchroNet*.

Conventions

Product Name

- In version 3.5, SOLID *Server* or SOLID Web Engine is now known as SOLID *Embedded Engine.* Note that this guide may still contain references to the old name SOLID *Server.*
- In this guide, "Solid server" or "Solid database" is used synonymously to refer to the server or database used in either SOLID products, SOLID *Embedded Engine* or SOLID *SynchroNet*.
- In this guide, "SOLID" used alone and in uppercase refers to both products, SOLID *SynchroNet* and SOLID *Embedded Engine*. In addition, "SOLID" is the short company name for Solid Information Technology (SOLID).

Typographic

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.

Other SOLID Documentation

SOLID documentation is distributed as printed material or in an electronic format (PDF, HTML, or Windows Help files).

SOLID Online Services on our Web server offer the latest product and technical information free of charge. The service is located at:

`http://www.solidtech.com/`

Electronic Documentation

- **Read Me** contains installation instructions and additional information about the specific product version. This `readme.txt` file is typically copied onto your system when you install the software.

- **Release Notes** contains additional information about the specific product version. This `relnotes.txt` file is typically copied onto your system when you install the software.
- **SOLID *SynchroNet* Guide** describes administrative procedures for SOLID *SynchroNet*. It also provides information about SOLID SQL functionality.
- **SOLID *Embedded Engine* Administrator Guide** describes administrative procedures for SOLID *Embedded Engine*, including tools and utilities, and also reference information.

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.
- Database Language — SQL: ANSI X3H2 and ISO/IEC JTC1/SC21/WG3 9075:1992 (SQL-92).
- X/Open CAE Specification, *Structured Query Language* (SQL), C201 (X/Open Company Ltd., U.K., 1992).

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

- Date, C. J, with Darwen, Hugh.: *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).
- Melton, Jim and Simon, Alan R.: *Understanding the new SQL: A Complete Guide* (Morgan Kaufmann, 1993).
- Pascal, Fabian: *SQL and Relational Basics* (M & T Books, 1990).
- Trimble, J. Harvey, Jr. and Chappell, David: *A Visual Introduction to SQL* (Wiley, 1989).
- Van der Lans, Rick F.: *Introduction to SQL* (Addison-Wesley, 1988).

- Vang, Soren: *SQL and Relational Databases* (Microtrend Books, 1990).
- Viescas, John: *Quick Reference Guide to SQL* (Microsoft Corp., 1989).

1

Introduction to SOLID APIs

This chapter provides an overview of the application programming interfaces available to you for accessing SOLID databases. These APIs include:

- SOLID *ODBC Driver*
- SOLID *Light Client*
- SOLID *JDBC Driver*

SOLID *ODBC Driver*

SOLID's 32-bit native *ODBC Driver* conforms to the Microsoft ODBC 3.5.x API standard. The SOLID *ODBC Driver* maintains a transaction for each active database connection. For differences in SOLID implementation, refer to the appropriate topic in this manual.

You can download the SOLID *ODBC Driver Package* as a part of the SDK from the SOLID Web site. For other environments that support the *ODBC Driver* as an option, see the SOLID Web site.

Depending on the applications request, the SOLID *ODBC Driver* can automatically commit each SQL statement or wait for an explicit commit or rollback request. When the driver performs a commit or rollback operation, the driver resets all statement requests associated with the connection.

The Driver Manager, which applies to Windows NT/2000/98/95 environments, manages the work of allowing an application to switch connections while transactions are in progress on the current connection.

Using SOLID *ODBC Driver* Functions

Users on all platforms can also access *ODBC Driver* supported functions with SOLID *ODBC API*. The SOLID *ODBC API* is the native call level interface (CLI) for SOLID databases. It is a DLL for Windows and a library for other environments. SOLID *ODBC API* is compliant with ANSI X3H2 SQL CLI.

SOLID's implementation of ODBC API supports a rich set of database access operations sufficient for creating robust database applications, including:

- Allocating and deallocating handles
- Getting and setting attributes
- Opening and closing database connections
- Accessing descriptors
- Executing SQL statements
- Accessing schema metadata
- Controlling transactions
- Accessing diagnostic information

ODBC API Basic Application Steps

A database application calls the SOLID *ODBC API* directly or through the ODBC Driver Manager, to perform all interactions with a database. These interfaces enable applications to establish multiple database connections simultaneously and to process multiple statements simultaneously.

An application using ODBC API performs the following tasks:

1. The application allocates memory for an environment handle (*henv*) and a connection handle (*hdbc*); both are required to establish a database connection.

   An application may request multiple connections for one or more data sources. Each connection is considered a separate transaction space.

2. The **SQLConnect** call establishes the database connection, specifying the server name, user id, and password.
3. The application then allocates memory for a statement handle and calls either **SQLExecDirect**, which both prepares and executes a SQL statement, or **SQLPrepare** and **SQLExecute**, which allows statements to be executed multiple times.
4. If the statement was a SELECT, the resulting columns need to be bound to variables in the application. This is done by using **SQLBindCol**. The rows can then be fetched using **SQLFetch** repeatedly. SELECT statements need to be committed, as soon as processing of the resultset is done.

5. If the statement was a UPDATE, DELETE or INSERT, the application needs to check if the execution succeeded and call **SQLTransact** to commit the transaction.
6. Finally the application closes the connection.

Read *Chapter 2,"Using SOLID ODBC API,"* for more information on using these APIs.

SOLID *Light Client*

SOLID *Light Client* allows you to develop small-footprint applications using C (or any tool that conforms to the C function call conversion). It is a 20-function subset of the ODBC API, providing full SQL capabilities for application developers accessing data from SOLID databases. It provides functions for controlling database connections, executing SQL statements, retrieving result sets, committing transactions, and other data management functionality. Read *Chapter 5,"Using SOLID Light Client,"*for more details.

SOLID *JDBC Driver*

SOLID *JDBC Driver* allows you to develop your application with a Java tool that accesses the database using JDBC. The JDBC API, the core API for JDK 1.2, defines Java classes to represent database connections, SQL statements, result sets, database metadata, etc. It allows you to issue SQL statements and process the results. JDBC is the primary API for database access in Java. Read *Chapter 6,"Using the SOLID JDBC Driver,"* for more details.

2

Using SOLID *ODBC API*

This chapter contains SOLID-specific information for developing applications with ODBC API. In general, SOLID conforms to the Microsoft ODBC 3.5.x standard. This chapter details those areas where SOLID-specific usage applies and where support for options, datatypes, and functions differ.

Note

This Programmer Guide does not contain a full ODBC API reference. This chapter provides SOLID-specific additions, supplements, and usage samples to that material.

For details on developing applications with ODBC API, refer to the Microsoft® Data Access SDK *Online ODBC Programmer's Reference*. For your convenience, the main portions of this reference are available in PDF format on the SOLID Web site. This reference includes usage chapters describing how to develop applications with ODBC API, as well as a comprehensive function reference.

Calling Functions

Programs that call standard Microsoft ODBC functions must include the SQL.H, SQLEXT.H header files. These files define ODBC constants and types and provide function prototypes for all standard ODBC functions. Functions defined in these header files provide support for ASCII character data types only.

Programs that call SOLID *ODBC API* specific functions must include the Microsoft ODBC standard header SQLUCODE.H and the Microsoft Visual C++ (or devstudio) package INCLUDE file, WCHAR.H. These files define constants and types and provide function prototypes for all SOLID *ODBC API* functions. Functions defined in these header files provide support for ASCII and Unicode character data types.

For details on driver, API, and SQL conformance levels, refer to the Microsoft ODBC API Specification (Part I PDF file), "Introduction to ODBC" available on the SOLID Web site.

Using the ODBC Driver Manager

In the Windows platform, the Driver Manager is a DLL to gain access to the SOLID *ODBC Driver*. An application typically links with the Driver Manager import library (ODBC.LIB) to gain access to the Driver Manager. In other platforms, SOLID provides the same driver library to be dynamically /statically linked to the application.

Note

Applications accessing ODBC API may bypass the Driver Manager to access data from SOLID databases by directly linking with the driver. The Driver Manager only applies to Windows NT/2000/98/95 environments. Other platforms do not use the Driver Manager; however, the Driver Manager is required if applications that connect to SOLID use OLE DB or ADO APIs or if database tools that require the Driver Manager, such as Microsoft Access, FoxPro, or Crystal Reports are to be used.

For basic application steps that occur whenever an application calls an ODBC function and details on calling ODBC functions, refer to the Microsoft ODBC API Specification (Part I PDF file), "Introduction to ODBC" available on the SOLID Web site.

Data Types

Appendix D, "Data Types" provides information about SOLID supported data types. The C standard Microsoft ODBC data types are defined in SQL.H and SQLEXT.H. The functions defined in these header files provide support for ASCII character string data types only.

Note

The C data types of SOLID *ODBC API* are defined in SQLUCODE.H and WCHAR.H. These files provide unicode format.

Scalar Functions

Scalar functions return a value for each row. For example, the absolute value scalar function takes a numeric column as an argument and returns the absolute value of each value in the column. For a list of functions that can be invoked with the following ODBC escape sequence, refer to *Appendix E, "Scalar Functions"*:

```
{fn scalar-function}
```

SOLID Native Scalar Functions

SOLID provides the following native scalar functions, which *cannot* be invoked using the ODBC escape sequence. They are:

- CURRENT_CATALOG() - returns WVARCHAR string. which contains the current active catalog name. This name is the same as ODBC scalar function {fn DATABASE()}.
- LOGIN_CATALOG() - returns WVARCHAR string, which contains the login catalog for the connected user (currently the login catalog is the same as the system catalog).
- CURRENT_SCHEMA() - returns WVARCHAR string, which contains the current active schema name.

Function Return Codes

When an application calls a 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.

Connecting to a Data Source

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.

For example, the section that describes the SOLID data source might be:

```
[soliddb]
DRIVER32=C:\WINNT\System32\bocw3235.dll
```

Notes

1. If the used data source name can be interpreted as a valid SOLID (server) network name, the client first connects using the information given in the data source name. A valid network name consists of a *communication protocol*, and optional *host computer name* and a *server name*. See the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide** for more information about listen names.

2. If the data source name is not a valid SOLID (server) listen name, the information needed to locate a server in the network is read from the ODBC.INI file or registry.

   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 in order to connect to the data source.

3. Applications that bypass the Driver Manager to access data from SOLID databases by directly linking with the driver must connect to the server using a valid listen name. If the data source name is not a valid SOLID (server) listen name, all SOLID client applications search for a valid listen name from:

   a) the SOLID.INI file
   b) the ODBC.INI or registry

   See **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide** for more information about the use of data source names.

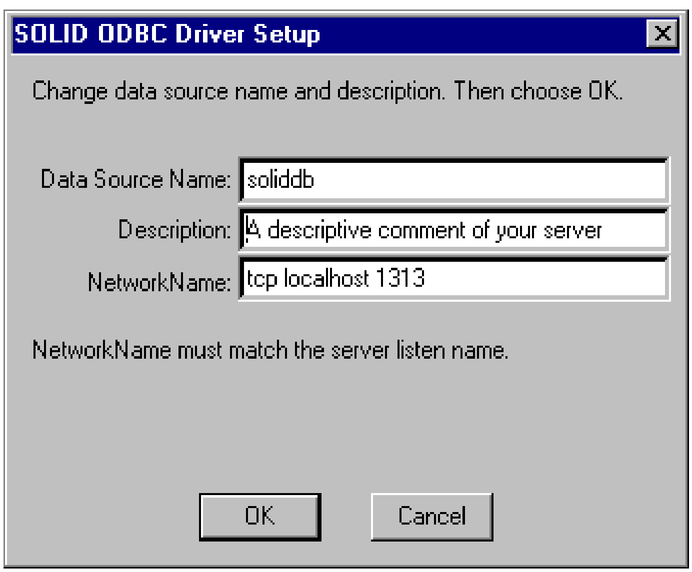
▶ **Note**

When an application uses ODBC API directly and calls **SQLConnect** and does not specify a SOLID *Embedded Engine or* SOLID *SynchroNet* network name, it is read from the parameter Connect in the [Com] section of the `solid.ini` file. The `solid.ini` file must reside in the current working directory of the application or in path specified by the SOLIDDIR environment variable.

Configuring the SOLID ODBC Data Source for Windows

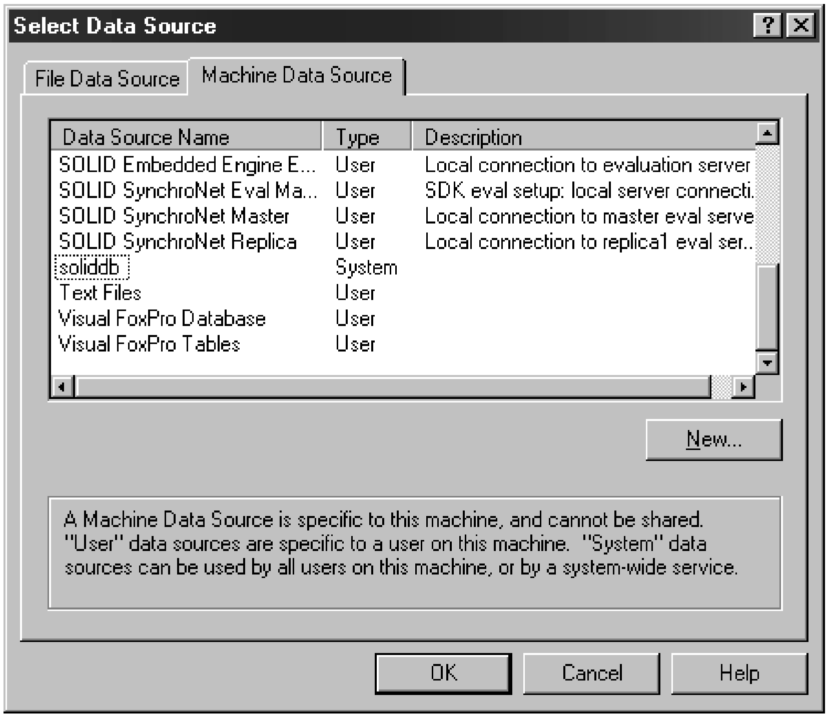
To configure an ODBC data source for Windows, users perform the following steps:

1. Invoke **ODBC32 Data Sources** from the Control Panel.
2. Select the SOLID ODBC 3.50 Driver.
3. Enter the Data Source configuration in the SOLID ODBC Driver Setup box as shown in the following example. Note that the NetworkName entry should be compliant with the database server listen addresses defined in `solid.ini`.

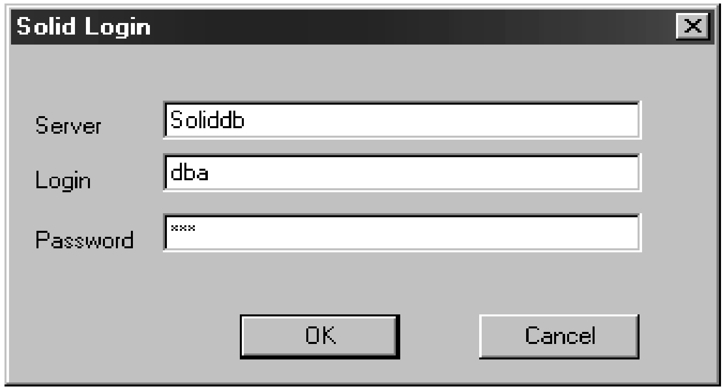


Retrieving User Login Information

If the application calls **SQLDriverConnect** and requests that the user be prompted for information, the Driver Manager displays a dialog box similar to the following example:



On request from the application, the driver retrieves login information by displaying the following dialog box:



Executing Transactions

In *auto-commit* mode, every SQL statement is a complete transaction, which is automatically committed when the next statement is executed. Please refer to the important note below on SELECT statements and autocommit mode.

In *manual-commit* mode, a transaction consists of one or more statements. In manual-commit mode, when an application submits a 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 **SQLEndTran**.

Cursors and Autocommit

! **Important**

Note that committing SELECT/read-only transactions is required in SOLID, even if you plan to use the AUTOCOMMIT ON mode.

If a transaction is not committed, it stays alive until the client disconnects or the transaction is timed out. This can result in a long-running transaction that can cause significant performance problems. SOLID saves the ’read-level’ of a transaction in memory. All subsequent transactions from other connections are also maintained in the memory. (This behavior is part of the advanced predicate validation and row versioning in the Bonsai Tree technology.)

Committing transactions keeps the amount of needed memory small. If a transaction is not committed, memory growth (due, for example, to a non-committed 'select transaction') may become large and exceed the available resources, eventually causing a performance problem.

AUTOCOMMIT mode set to "on" amplifies this issue because SELECTs in AUTOCOMMIT mode are committed automatically only when the next statement is executed. To prevent this problem from occurring, users should explicitly close the cursor, which allows for the commit to occur and prevents unwarranted Bonsai Tree growth.

Setting SOLID Parameter Values

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 **SQLFree-Handle or SQLFreeStmt** with the SQL_RESET_PARAMS 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.

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**.

Executing Functions Asynchronously

Note

ODBC drivers for SOLID *Embedded Engine* or *SOLID SynchroNet* do not support asynchronous execution.

Using ODBC Extensions to SQL

ODBC defines extensions to SQL, which are common to most DBMS's. For details on SQL extensions, refer to "Escape Sequences in ODBC" in the Microsoft ODBC API Specification (Part I PDF file that is available on the SOLID Web site) which contains the introductory part of the Microsoft *ODBC Programmer's Reference*.

Included in the ODBC extensions to SQL are:

- Procedures
- Hints

Details on SOLID usage for these extensions are described in the following sections.

Procedures

Stored procedures are procedural program code containing typically a single or several SQL statements and program logic. They are stored in the database and executed with one call from the application or another stored procedure. Read *"Stored Procedures"* on page 3-1 for a full description of SOLID stored procedures.

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

```
{[?=] 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 through the optional parameter marker **?=** shown in the syntax above. For input and input/output parameters, *parameter* can be a literal or a parameter marker. Because some data sources do not accept literal parameter values, be sure that interoperable applications use parameter markers. For output parameters, *parameter* must be a parameter marker. 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.

Procedure calls do not require input and input/output parameters. Note the following rules:

- A procedure called with parentheses but with parameters omitted, such as {call *procedure_name*()}, may cause the procedure to fail.
- A procedure called without parentheses, such as {call *procedure_name*}, returns no parameter values.
- When a parameter is omitted, the comma delimiting it from other parameters must be present.
- Omitted input or input/output parameters cause the driver to instruct the data source to use the default value of the parameter. As an option, a parameters default value can be set using the value of the length/indicator buffer bound to the parameter to SQL_DEFAULT_PARAM.
- Omitted input/output parameters or literal parameter values cause the driver to discard the output value.
- Omitted parameter markers for a procedure's return value cause the driver to discard the return value.
- If an application specifies a return value parameter for a procedure that does not return a value, the driver sets the value of the length/indicator buffer bound to the parameter to SQL_NULL_DATA.

To determine if a data source supports procedures, an application calls **SQLGetInfo** with the SQL_PROCEDURES information type. For more information about procedures, read *"Stored Procedures"* on page 3-1.

Hints

Within a query, Optimizer directives or *hints* can be specified to determine the query execution plan that is used. Hints are detected through a pseudo comment syntax from SQL2. SOLID provides its own extensions to hints:

```
--(* vendor (SOLID), product (Engine), option(hint)
--hint *)--
```

hint :=

```
[MERGE JOIN |
LOOP JOIN |
JOIN ORDER FIXED |
```

```
   INTERNAL SORT |
   EXTERNAL SORT |
   INDEX [REVERSE] table_name.index_name |
   PRIMARY KEY [REVERSE] table_name
   FULL SCAN table_name |
  [NO] SORT BEFORE GROUP BY]
```

The pseudo comment prefix is followed by identifying information. Vendor is specified as **SOLID**, product as **Engine**, and the option, which is the pseudo comment class name, as a valid hint.

Hints always follow the SELECT, UPDATE, or DELETE keyword that applies to it.

Note

Hints are not allowed after the INSERT keyword.

Each subselect requires its own hint; for example, the following are valid uses of hints syntax:

INSERT INTO ... SELECT *hint* FROM ...

UPDATE *hint* TABLE ... WHERE *column* = (SELECT *hint* ... FROM ...)

DELETE *hint* TABLE ... WHERE *column* = (SELECT hint ... FROM ...)

Example 1

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--MERGE JOIN
--JOIN ORDER FIXED *)--
*
FROM TAB1 A, TAB2 B;
WHERE A.INTF = B.INTF;
```

Example 2

```
SELECT
--(* vendor(SOLID), product(Engine), option(hint)
--INDEX TAB1.INDEX1
```

```
--INDEX TAB1.INDEX1 FULL SCAN TAB2 *)--
*
FROM TAB1, TAB2
WHERE TAB1.INTF = TAB2.INTF;
```

Hint is a specific semantic, corresponding to a specific behavior. Following is a list of SOLID-supported hints:

Hint	Definition
MERGE JOIN	Directs the Optimizer to choose the merge join access plan in a select query for all tables listed in the FROM clause. Use this hint when the data is sorted by a join key and the nested loop join performance is not adequate. The MERGE JOIN option selects the merge join only where there is an equal predicate between tables. Otherwise, the Optimizer selects LOOP JOIN even if the MERGE JOIN hint is specified. Note that when data is not sorted before performing the merge operation, the SOLID query executor sorts the data. When considering the usage of this hint, keep in mind that the merge join with a sort is more resource intensive than the merge join without the sort.
LOOP JOIN	Directs the Optimizer to pick the nested loop join in a select query for all tables listed in the FROM clause. By default, the Optimizer does not pick the nested loop join. Using the loop join when tables are small and fit in memory may offer greater efficiency than using other complex join algorithms.
JOIN ORDER FIXED	Specifies that the Optimizer use tables in a join in the order listed in the FROM clause of the query. This means that the Optimizer does not attempt to rearrange any join order and does not try to find alternate access paths to complete the join. Before using this hint, be sure to run the EXPLAIN PLAN to view the associated plan. This gives you an idea on the access plan used for executing the query with this join order.
INTERNAL SORT	Specifies that the query executor use the internal sort. Use this hint if the expected result set is small (100s of rows as opposed to 1000s of rows); for example, if you are performing some aggregates, ORDER BY with small result sets, or GROUP BY with small result sets, etc. This hint avoids the use of the more expensive external sort.

Hint	Definition
EXTERNAL SORT	Specifies that the query executor use the external sort. Use this hint when the expected result set is large and does not fit in memory; for example, if the expected result set has 1000s of rows. In addition, specify the SORT working directory in the `solid.ini` before using the external sort hint. If a working directory is not specified, you will receive a run-time error.
INDEX [REVERSE] *table_name.index_name*	Forces a given index scan for a given table. In this case, the Optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query. Before using this hint, it is recommended that you run the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query. The optional keyword **REVERSE** returns the rows in the reverse order. In this case, the query executor begins with the last page of the index and starts returning the rows in the descending (reverse) key order of the index. Note that in *tablename.indexname*, the *tablename* is a fully qualified table name which includes the *catalogname* and *schemaname*.
PRIMARY KEY [REVERSE] *tablename*	Forces a primary key scan for a given table. The optional keyword **REVERSE** returns the rows in the reverse order. If the primary KEY is not available for the given table, then you will receive a run-time error.
FULL SCAN *table_name*	Forces a table scan for a given table. In this case, the optimizer does not proceed to evaluate if there are any other indexes that can be used to build the access plan or whether a table scan is better for the given query. Before using this hint, it is recommended that you run the EXPLAIN PLAN output to ensure that the plan generated is optimal for the given query. In this FULL SCAN, the query executor tries to use the PRIMARY KEY, if one is available. If not, then it uses the SYSTEM KEY.

Hint	Definition
[NO] SORT BEFORE GROUP BY	Indicates whether the SORT operation occurs before the result set is grouped by the GROUP BY columns. If the grouped items are few (100s of rows) then use NO SORT BEFORE. On the other hand, if the grouped items are large (1000s of rows), then use SORT BEFORE.

For more examples on hints, refer to the "Performance Tuning" chapter in the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide**

Additional Extension Functions

ODBC provides the following functions related to SQL statements. Refer to the Microsoft ODBC API Specification (Part II PDF file that is available on the SOLID Web site) for more information about these functions.

Function	Description
SQLDescribeParam	Retrieves information about prepared parameters.
SQLNumParams	Retrieves the number of parameters in a SQL statement.
SQLSetStmtAttr SQLSetConnectAttr SQLGetStmtAttr	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 **SQLSetConnectAttr** sets options for all statements in a connection.

Using Cursors

The ODBC Driver uses a cursor concept to keep track of its position in the resultset, that is, in the data rows retrieved from the database. A cursor is used for tracking and indicating the current position, similarly as the cursor on a CRT screen indicates cursor position.

Each time an application calls **SQLFetch**, the driver moves the cursor to the next row and returns that row. The cursor supported by the core ODBC functions only scrolls forward, one row at a time. (To re-retrieve a row of data that it has already retrieved from the result set, the application must close the cursor by calling **SQLFreeStmt** with the SQL_CLOSE option, re-execute the **SELECT** statement, and fetch rows with **SQLFetch** until the target row is retrieved.)

Assigning Storage for Rowsets (Binding)

In addition to binding individual rows of data, an application can call **SQLBindCol** to assign storage for a *rowset* (one or more rows of data). By default, rowsets are bound in column-wise fashion. They can also be bound in row-wise fashion.

To specify how many rows of data are in a rowset, an application calls **SQLSetStmtAttr** with the SQL_ROWSET_SIZE option.

Column-Wise Binding

To assign storage for column-wise bound results, an application performs the following steps for each column to be bound:

1. Allocates an array of data storage buffers. The array has as many elements as there are rows in the rowset.
2. Allocates an array of storage buffers to hold the number of bytes available to return for each data value. The array has as many elements as there are rows in the rowset.
3. Calls **SQLBindCol** and specifies the address of the data array, the size of one element of the data array, the address of the number-of-bytes array, and the type to which the data will be converted. When data is retrieved, the driver will use the array element size to determine where to store successive rows of data in the array.

Row-Wise Binding

To assign storage for row-wise bound results, an application performs the following steps:

1. Declares a structure that can hold a single row of retrieved data and the associated data lengths. (For each column to be bound, the structure contains one field to contain data and one field to contain the number of bytes of data available to return.)
2. Allocates an array of these structures. This array has as many elements as there are rows in the rowset.
3. Calls **SQLBindCol** for each column to be bound. In each call, the application specifies the address of the column's data field in the first array element, the size of the data field, the address of the column's number-of-bytes field in the first array element, and the type to which the data will be converted.
4. Calls **SQLSetStmtAttr** with the SQL_BIND_TYPE option and specifies the size of the structure. When the data is retrieved, the driver will use the structure size to determine where to store successive rows of data in the array.

Cursor Support

Applications require different means to sense changes in the tables underlying a result set. For example, when balancing financial data, an accountant needs data that appears static; it is impossible to balance books when the data is continually changing. When selling concert tickets, a clerk needs up-to-the minute, or dynamic, data on which tickets are still available. Various cursor models are designed to meet these needs, each of which requires different sensitivities to changes in the tables underlying the result set.

SOLID cursors which are set with **SQLSetStmtAttr** as "dynamic" closely resemble static cursors, with some dynamic behavior. SOLID dynamic cursor behavior is static in the sense that changes made to the resultset by other users are not visible to the user, as opposed to dynamic cursors in which changes are visible to the user.

The exception in SOLID's cursor behavior is that transactions are able to view their own data changes, but cannot view the changes made by other transactions. The conditions in SOLID, however, that cause a user's own changes to be invisible to that user are:

- In a SELECT statement when an ORDER BY clause or a GROUP BY clause is used, SOLID caches the result set, which causes the user's own change to be invisible to the user.
- In applications written using ADO or OLE DB, SOLID cursors are more like dynamic ODBC cursors to enable functions such as a row set update.

Specifying the Cursor Type

To specify the cursor type, an application calls **SQLSetStmtAttr** with the SQL_CURSOR_TYPE option. The application can specify a cursor that only scrolls forward, a static cursor, or a dynamic cursor.

Unless the cursor is a forward-only cursor, an application calls **SQLExtendedFetch** (ODBC 2.x) or **SQLFetchScroll** (ODBC 3.x) to scroll the cursor backwards or forwards.

Cursor Support

Three types of cursors are defined in ODBC 3.51:

- Driver Manager supported cursors
- Server supported cursors
- Driver supported cursors

SOLID cursors are server supported cursors.

Cursors and Autocommit

For SOLID-specific information on cursors and autocommit, read *"Setting SOLID Parameter Values"* on page 2-8.

Specifying Cursor Concurrency

Concurrency is the ability of more than one user to use the same data at the same time. A transaction is *serializable* if it is performed in a manner in which it appears as if no other transactions operate on the same data at the same time. For example, assume one transaction doubles data values and another adds 1 to data values. If the transactions are serializable and both attempt to operate on the values 0 and 10 at the same time, the final values will be 1 and 21 or 2 and 22, depending on which transaction is performed first. If the transactions are not serializable, the final values will be 1 and 21, 2 and 22, 1 and 22, or 2 and 21; the sets of values 1 and 22, and 2 and 21, are the result of the transactions acting on each value in a different order.

Serializability is considered necessary to maintain database integrity. For cursors, it is most easily implemented at the expense of concurrency by locking the result set. A compromise between serializability and concurrency is *optimistic concurrency control*. In a cursor using optimistic concurrency control, the driver does not lock rows when it retrieves them. When the application requests an update or delete operation, the driver or data source checks if the row has changed. If the row has not changed, the driver or data source prevents other transactions from changing the row until the operation is complete. If the row has changed, the transaction containing the update or delete operation fails.

Using Bookmarks

A bookmark is a 32-bit value that an application uses to return to a row. SOLID provides no support for bookmarks.

Error Text Format

Error messages returned by **SQLError** come from two sources: data sources and components in an ODBC connection. Typically, data sources do not directly support ODBC. Consequently, if a component in an ODBC connection receives an error message from a data source, it must identify the data source as the source of the error. It must also identify itself as the component that received the error.

If the source of an error is the component itself, the error message must explain this. Therefore, the error text returned by **SQLError** has two different formats: one for errors that occur in a data source and one for errors that occur in other components in an ODBC connection.

For errors that do not occur in a data source, the error text must use the format:

[*vendor_identifier***][***ODBC_component_identifier***]**

component_supplied_text

For errors that occur in a data source, the error text must use the format:

[*vendor_identifier***][***ODBC_component_identifier***]**

[*data_source_identifier***]** *data_source_supplied_text*

The following table shows the meaning of each element.

Element	Meaning
vendor_identifier	Identifies the vendor of the component in which the error occurred or that received the error directly from the data source.
ODBC_component_identifier	Identifies the component in which the error occurred or that received the error directly from the data source.
data_source_identifier	Identifies the data source. For single-tier drivers, this is typically a file format. For multiple-tier drivers, this is the DBMS product.
component_supplied_text	Generated by the ODBC component.
data_source_supplied_text	Generated by the data source.

Note

The brackets (**[ ]**) are included in the error text; they do not indicate optional items.

Sample Error Messages

The following examples show how various components in an ODBC connection might generate the text of error messages and how SOLID returns them to the application with **SQLError**.

01000	General warning
01S00	Invalid connection string attribute

08001	Client unable to establish connection

SQLSTATE values are strings that contain five characters; the first two are a string class value, followed by a three-character subclass value. For example **01000** has **01** as its class value and **000** as its subclass value. Note that a subclass value of 000 means there is no subclass for that SQLSTATE. Class and subclass values are defined in SQL-92.

Class value	Meaning
01	Indicates a warning and includes a return code of SQL_SUCCESS_WITH_INFO.
01, 07, 08, 21, 22, 25, 28, 34, 3C, 3D, 3F, 40, 42, 44, HY	Indicates an error that includes a return value of SQL_ERROR.
IM	Indicates warning and errors that are derived from ODBC.

Processing Error Messages

Applications provide users with all the error information available through **SQLError**: the ODBC SQLSTATE, the native error code, the error text, and the source of the error. The application may parse the error text to separate the text from the information identifying the source of the error. It is the application's responsibility to take appropriate action based on the error or provide the user with a choice of actions.

The ODBC interface provides functions that terminate statements, transactions, and connections, and free statement, connection, and environment handles.

Terminating Transactions and Connections

The ODBC interface provides functions that terminate statements, transactions, and connections, and free statement (*hstmt*), connection (*hdbc*), and environment (*henv*) handles.

Terminating Statement Processing

To free resources associated with a statement handle, an application calls **SQLFreeStmt** with the following options:

- **SQL_CLOSE** - Closes the cursor, if one exists, and discards pending results. The application can use the statement handle again later. In ODBC 3.5.x, **SQLCloseCursor** can also be used.
- **SQL_UNBIND** - Frees all return buffers bound by **SQLBindCol** for the statement handle.

- **SQL_RESET_PARAMS** - Frees all parameter buffers requested by **SQLBindParameter** for the statement handle.

The **SQLFreeHandle** is used to close the cursor if one exists, discard pending results, and free all resources associated with the statement handle.

Terminating Transactions

An application calls **SQLTransact** to commit or roll back the current transaction.

Terminating Connections

To terminate a connection to a driver and data source, an application performs the following steps:

1. Calls **SQLDisconnect** to close the connection. The application can then use the handle to reconnect to the same data source or to a different data source.
2. Calls **SQLFreeHandle** to free the connection or environment handle and free all resources associated with the handle.

Constructing an Application

This section provides two examples of C-language source code for applications.

Sample Application Code

The following sections contain two examples that are written in the C programming language:

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

This example can use either Microsoft ODBC header files for ASCII data or SOLID ODBC API header files for unicode data.

Static SQL Example

The following example constructs SQL statements within the application.

```
#if (defined(SS_UNIX) || defined(SS_LINUX))
#include <sqlunix.h>
#else
```

```
#include <windows.h>
#endif


#if SOLIDODBCAPI
#include <sqlucode.h>
#include <wchar.h>
#else
#include <sql.h>
#include <sqlext.h>
#endif

#include <stdio.h>
#include <assert.h>

#define MAX_NAME_LEN 50
#define MAX_STMT_LEN 100

/***********************************************************************
          Function Name: PrintError
          Purpose: To Display the error associted with the handle
***********************************************************************/
SQLINTEGER PrintError(SQLSMALLINT handleType,SQLHANDLE handle)
{
   SQLRETURN     rc = SQL_ERROR;
   SQLWCHAR      sqlState[6];
   SQLWCHAR      eMsg[SQL_MAX_MESSAGE_LENGTH];
   SQLINTEGER    nError;

   rc = SQLGetDiagRecW(handleType,handle,1,(SQLWCHAR *)&sqlState,
                 (SQLINTEGER *)&nError,(SQLWCHAR*)&eMsg,255,NULL);
   if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)   {
   printf("\n\t Error:%ls\n",eMsg);
```

```
    }
    return(SQL_ERROR);
}
/*********************************************************************
          Function Name: DrawLine
          Purpose: To Draw a specified charcter(chr) for specified
          number of times(len)
*********************************************************************/
void DrawLine(SQLINTEGER len,SQLCHAR chr)
{
    printf("\n");
    while(len > 0){
          printf("%c",chr);
          len--;
    }
    printf("\n");

}
/*********************************************************************
          Function Name: example1
          Purpose: Connect to the specified data source and execute the
          set of SQL Statements
*********************************************************************/
SQLINTEGER example1(SQLCHAR *server, SQLCHAR *uid, SQLCHAR *pwd)
{
    SQLHENV       henv;
    SQLHDBC       hdbc;
    SQLHSTMT      hstmt;
    SQLRETURN     rc;

    SQLINTEGER    id;
    SQLWCHAR      drop[MAX_STMT_LEN];
    SQLCHAR       name[MAX_NAME_LEN+1];
```

```
    SQLWCHAR      create[MAX_STMT_LEN];
    SQLWCHAR      insert[MAX_STMT_LEN];
    SQLWCHAR      select[MAX_STMT_LEN];
    SQLINTEGER    namelen;


          /* Allocate environment and connection handles. */
          /* Connect to the data source. */
          /* Allocate a statement handle. */

          rc = SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&henv);
          if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                return(PrintError(SQL_HANDLE_ENV,henv));

          rc =
SQLSetEnvAttr(henv,SQL_ATTR_ODBC_VERSION,(SQLPOINTER)SQL_OV_ODBC3,SQL_NT
S);
          if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                return(PrintError(SQL_HANDLE_ENV,henv));

          rc = SQLAllocHandle(SQL_HANDLE_DBC,henv,&hdbc);
          if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                return(PrintError(SQL_HANDLE_ENV,henv));

          rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS,pwd,
SQL_NTS);
          if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                return(PrintError(SQL_HANDLE_DBC,hdbc));

          rc = SQLAllocHandle(SQL_HANDLE_STMT,hdbc,&hstmt);
          if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                return(PrintError(SQL_HANDLE_DBC,hdbc));
```

```
/* drop table 'nameid' if exists, else continue*/
wcscpy(drop,L"DROP TABLE NAMEID");
printf("\n%ls",drop);
DrawLine(wcslen(drop),'-');

rc = SQLExecDirectW(hstmt,drop,SQL_NTS);
if (rc == SQL_ERROR)
       PrintError(SQL_HANDLE_STMT,hstmt);

/* commit work*/
rc = SQLEndTran(SQL_HANDLE_DBC,hdbc,SQL_COMMIT);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_DBC,hdbc));

/* create the table nameid(id integer,name varchar(50))*/
wcscpy(create,L"CREATE TABLE NAMEID(ID INT,NAME
VARCHAR(50))");
printf("\n%ls",create);
DrawLine(wcslen(create),'-');

rc = SQLExecDirectW(hstmt,create,SQL_NTS);
if (rc == SQL_ERROR)
       return(PrintError(SQL_HANDLE_STMT,hstmt));

/* commit work*/
rc = SQLEndTran(SQL_HANDLE_DBC,hdbc,SQL_COMMIT);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_DBC,hdbc));


/* insert data through parameters*/
wcscpy(insert,L"INSERT INTO NAMEID VALUES(?,?)");
printf("\n%ls",insert);
```

```
          DrawLine(wcslen(insert),'-');

          rc = SQLPrepareW(hstmt,insert,SQL_NTS);
          if (rc == SQL_ERROR)
                return(PrintError(SQL_HANDLE_STMT,hstmt));

          /* integer(id) data binding*/
          rc =
SQLBindParameter(hstmt,1,SQL_PARAM_INPUT,SQL_C_LONG,SQL_INTEGER,

0,0,&id,0,NULL);
          if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                return(PrintError(SQL_HANDLE_DBC,hdbc));


          /* char(name) data binding*/
          rc =
SQLBindParameter(hstmt,2,SQL_PARAM_INPUT,SQL_C_CHAR,SQL_VARCHAR,

0,0,&name,sizeof(name),NULL);
          if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                return(PrintError(SQL_HANDLE_DBC,hdbc));

          id = 100;
          strcpy(name,"SOLID");

          rc = SQLExecute(hstmt);
          if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                return(PrintError(SQL_HANDLE_DBC,hdbc));

          /* commit work*/
          rc = SQLEndTran(SQL_HANDLE_DBC,hdbc,SQL_COMMIT);
          if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                return(PrintError(SQL_HANDLE_DBC,hdbc));
```

```
/* free the statement buffers*/
rc = SQLFreeStmt(hstmt,SQL_RESET_PARAMS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_STMT,hstmt));

rc = SQLFreeStmt(hstmt,SQL_CLOSE);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_STMT,hstmt));

/* select data from the table nameid*/
wcscpy(select,L"SELECT * FROM NAMEID");
printf("\n%ls",select);
DrawLine(wcslen(select),'-');

rc = SQLExecDirectW(hstmt,select,SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_DBC,hdbc));

/* bind buffers for output data*/
id = 0;
strcpy(name,"");

rc = SQLBindCol(hstmt,1,SQL_C_LONG,&id,0,NULL);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_DBC,hdbc));

rc =
SQLBindCol(hstmt,2,SQL_C_CHAR,&name,sizeof(name),&namelen);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_DBC,hdbc));

rc = SQLFetch(hstmt);
```

```
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_DBC,hdbc));

printf("\n Data ID:%d",id);
printf("\n Data Name:%s(%d)\n",name,namelen);

rc = SQLFetch(hstmt);
assert(rc == SQL_NO_DATA);


/* free the statement buffers*/
rc = SQLFreeStmt(hstmt,SQL_UNBIND);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_STMT,hstmt));

rc = SQLFreeStmt(hstmt,SQL_CLOSE);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_STMT,hstmt));

/* Free the statement handle. */
rc = SQLFreeHandle(SQL_HANDLE_STMT,hstmt);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_STMT,hstmt));

/* Disconnect from the data source. */
rc = SQLDisconnect(hdbc);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_DBC,hdbc));

/* Free the connection handle. */
rc = SQLFreeHandle(SQL_HANDLE_DBC,hdbc);
       if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_DBC,hdbc));
```

```
        /* Free the environment handle. */
        rc = SQLFreeHandle(SQL_HANDLE_ENV,henv);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
               return(PrintError(SQL_HANDLE_ENV,henv));

        return(0);

}
```

Interactive Ad Hoc Query Example

The following example illustrates how an application can determine the nature of the result set prior to retrieving results.

```
#if (defined(SS_UNIX) || defined(SS_LINUX))
#include <sqlunix.h>
#else
#include <windows.h>
#endif

#if SOLIDODBCAPI
#include <sqlucode.h>
#include <wchar.h>
#else
#include <sql.h>
#include <sqlext.h>
#endif

#include <stdio.h>

#ifndef TRUE
#define TRUE 1
#endif
```

```
#define MAXCOLS 100
#define MAX_DATA_LEN 255

SQLHENV    henv;
SQLHDBC    hdbc;
SQLHSTMT   hstmt;

/*************************************************************************
   Function Name : PrintError
   Purpose       : To Display the error associted with the handle
*************************************************************************/
SQLINTEGER PrintError(SQLSMALLINT handleType,SQLHANDLE handle)
{
   SQLRETURN     rc = SQL_ERROR;
   SQLCHAR       sqlState[6];
   SQLCHAR       eMsg[SQL_MAX_MESSAGE_LENGTH];
   SQLINTEGER    nError;

   rc = SQLGetDiagRec(handleType,handle,1,(SQLCHAR *)&sqlState,
           (SQLINTEGER *)&nError,(SQLCHAR *)&eMsg,255,NULL);
   if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)   {

   printf("\n\t Error:%s\n",eMsg);
   }
   return(SQL_ERROR);
}
/*************************************************************************
   Function Name        : DrawLine
   Purpose              : To Draw a specified charcter(line) for
   specified number of times(len)
*************************************************************************/
void DrawLine(SQLINTEGER len,SQLCHAR line)
{
```

```
    printf("\n");
    while(len > 0){
            printf("%c",line);
            len--;
    }
    printf("\n");

}
/*************************************************************************
    Function Name : example2
    Purpose       : Connect to the specified data source and
    execute the given SQL statement According to the statement judge the
    result set
*************************************************************************/
SQLINTEGER example2(SQLCHAR *sqlstr)
{
    SQLINTEGERi;

    SQLCHAR       colname[32];
    SQLSMALLINT   coltype;
    SQLSMALLINT   colnamelen;
    SQLSMALLINT   nullable;
    SQLINTEGER    collen[MAXCOLS];
    SQLSMALLINT   scale;
    SQLINTEGER    outlen[MAXCOLS];
    SQLCHAR       data[MAXCOLS][MAX_DATA_LEN];
    SQLSMALLINT   nresultcols;
    SQLINTEGER    rowcount,nRowCount=0,lineLength=0;
    SQLRETURN     rc;


            printf("\n%s",sqlstr);
            DrawLine(strlen(sqlstr),'=');
```

```
/* Execute the SQL statement. */
rc = SQLExecDirect(hstmt, sqlstr, SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_STMT,hstmt));


/* See what kind of statement it was. If there are */
/* no result columns, the statement is not a SELECT */
/* statement. If the number of affected rows is */
/* greater than 0, the statement was probably an */
/* UPDATE, INSERT, or DELETE statement, so print */
/* the number of affected rows. If the number of */
/* affected rows is 0, the statement is probably a */
/* DDL statement, so print that the operation was */
/* successful and commit it. */

rc = SQLNumResultCols(hstmt, &nresultcols);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
       return(PrintError(SQL_HANDLE_STMT,hstmt));

if (nresultcols == 0) {
       rc = SQLRowCount(hstmt, &rowcount);
       if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
              return(PrintError(SQL_HANDLE_STMT,hstmt));

       if (rowcount > 0 ) {
              printf("%ld rows affected.\n", rowcount);
       }
       else {
       printf("Operation successful.\n");
       }
```

```
            rc = SQLEndTran(SQL_HANDLE_DBC,hdbc,SQL_COMMIT);
            if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
                  return(PrintError(SQL_HANDLE_DBC,hdbc));
      }
      /* Otherwise, display the column names of the result */
      /* set and use the display_size() function to */
      /* compute the length needed by each data type. */
      /* Next, bind the columns and specify all data will */
      /* be converted to char. Finally, fetch and print */
      /* each row, printing truncation messages as */
      /* necessary. */

      else {
            for (i = 0; i < nresultcols; i++) {
                  rc = SQLDescribeCol(hstmt, i + 1,
colname,(SQLSMALLINT)sizeof(colname), &colnamelen,
                              &coltype, &collen[i],
&scale,&nullable);
                  if (rc != SQL_SUCCESS && rc !=
SQL_SUCCESS_WITH_INFO)

return(PrintError(SQL_HANDLE_STMT,hstmt));

                  printf("%s\t",colname);/* print column names*/

                  rc = SQLBindCol(hstmt, i + 1,
SQL_C_CHAR,data[i],sizeof(data[i]),&outlen[i]);
                  if (rc != SQL_SUCCESS && rc !=
SQL_SUCCESS_WITH_INFO)

return(PrintError(SQL_HANDLE_STMT,hstmt));

                  lineLength+=6+strlen(colname);
            }
```

```
                    DrawLine(lineLength-6,'-');

                     while (TRUE){

                   rc = SQLFetch(hstmt);
                          if (rc ==SQL_SUCCESS || rc ==
SQL_SUCCESS_WITH_INFO) {
                                 nRowCount++;
                                 for (i = 0; i < nresultcols; i++) {
                                        if (outlen[i] == SQL_NULL_DATA) {
                                               strcpy((char *)data[i],
"NULL");
                                 }
                                        printf("%s\t",data[i]);
                                 }
                                 printf("\n");
                          }

                          else {
                      if (rc == SQL_ERROR)
                          PrintError(SQL_HANDLE_STMT,hstmt);
                                 break;
                          }
                    }
                     printf("\n\tTotal Rows:%d\n",nRowCount);
             }

             SQLFreeStmt(hstmt,SQL_UNBIND);
             SQLFreeStmt(hstmt,SQL_CLOSE);
             return(0);

      }
```

Testing and Debugging an Application

The Microsoft ODBC SDK provides the following tools for application 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.
- Sample applications, including source code and makefiles.
  - A **#define**, ODBCVER, to specify which version of ODBC you want to compile your application with. To use the ODBC 3.5 constants and prototypes, add the following line to your application code before providing the include files.

    ```
    #define ODBCVER 0X0352
    ```

  - For ASCII data, use the following standard Microsoft include files:

    SQL.H and SQLEXT.H

  - For Unicode data, use the following Microsoft include files:

    SQLUCODE.H and WCHAR.H

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

Installing and Configuring ODBC Software

Users install ODBC software with a driver-specific setup program (built with the Driver Setup Toolkit that is shipped with the ODBC SDK) or an application-specific setup program. They configure the ODBC environment with the ODBC Administrator (also shipped with the ODBC SDK) or an application-specific administration program. Application developers must decide whether to redistribute these programs or write their own setup and administration programs. For more information about the Driver Setup Toolkit and the ODBC Administrator, see the *Microsoft ODBC SDK Guide* on the Microsoft Web site.

A setup program written by an application developer uses the installer DLL to retrieve information from the ODBC.INF file, which is created by a driver developer and describes the disks on which the ODBC software is shipped. The setup program also uses the installer DLL to retrieve the target directories for the Driver Manager and the drivers, record information about the installed drivers, and install ODBC software.

Administration programs written by application developers use the installer DLL to retrieve information about the available drivers, to specify default drivers, and to configure data sources.

Application developers who write their own setup and administration programs must ship the installer DLL and the ODBC.INF file.

With the current version of ODBC 3.5.x, the Installer for Windows does not contain the Microsoft Driver Manager. To maintain compatibility with ADO, OLE DB, and ODBC, Microsoft recommends obtaining the Driver Manager and installing it. To do this, users need to download the executable `mdac_typ.exe` from the Microsoft Web site and install it; this executable provides users with Driver Manager 3.5 or above. For the URL to the Microsoft Web site where this executable is found, refer to the SOLID Web site or the Release Notes.

3

Stored Procedures, Events, Triggers, and Sequences

In SOLID, a number of features are available that make it possible to move parts of the application logic into the database. These features include:

- stored procedures
- event alerts
- triggers
- sequences

Stored Procedures

Stored procedures are simple programs, or procedures, that are executed in Solid databases. The user can create procedures that contain several SQL statements or whole transactions, and execute them with single call statement. In addition to SQL statements, 3GL type control structures can be used enabling procedural control. In this way complex, data-bound transactions may be run on the server itself, thus reducing network traffic.

Granting execute rights on a stored procedure automatically invokes the necessary access rights to all database objects used in the procedure. Therefore, administering database access rights may be greatly simplified by allowing access to critical data through procedures.

This section explains in detail how to use stored procedures. In the beginning of this section the general concepts of using the procedures are explained. Later sections go more in-depth and describe the actual syntax of different statements in the procedures. The end of this section discusses transaction management, sequences and other advanced stored procedure features.

Basic procedure structure

A stored procedure is a standard SOLID database object that can be manipulated using standard DDL statements CREATE and DROP.

In its simplest form a stored procedure definition looks like:

"CREATE PROCEDURE *procedure_name*
parameter_section
BEGIN
declare_section_local_variables
procedure_body
END";

Note

Because SOLID *DBConsole* is not able to parse these statements, the whole statement must be enclosed in double quotes.

The following example creates a procedure called TEST:

```
"CREATE PROCEDURE test
BEGIN
END";
```

Procedures can be run by issuing a CALL statement followed by the name of the procedure to be invoked:

```
CALL test;
```

Naming Procedures

Procedure names have to be unique within a database schema.

All the standard naming restrictions considering database objects, like using reserved words, identifier lengths etc., apply to stored procedure names. For an overview and complete list of reserved keywords, see the appendix, "Reserved Words" in the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide**.

Parameter Section

A stored procedure communicates with the calling program using parameters. Stored procedures accept two types of parameters:

- Input parameters; given as an input to the procedure can be used inside the procedure.
- Output parameters; returned values from the procedure. Stored procedures may return a result set of several rows with output parameters as the columns.

The types of parameters must be declared. For supported data types, see the appendix, "Data Types" in the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide**.

The syntax used in parameter declaration is:

parameter_name parameter_datatype

Input parameters are declared between parentheses directly after the procedure name, output parameters are declared in a special RETURNS section of the procedure definition:

```
"CREATE PROCEDURE procedure_name
[ (input_param1 datatype,
input_param2 datatype, … >) ]
[ RETURNS
(output_param1 datatype,
 output_param2 datatype, … >) ]
BEGIN
END";
```

There can be any number of input and output parameters. Input parameters have to be supplied in the same order as they are defined when the procedure is called.

Declaring input parameters in the procedure heading make their values accessible inside the procedure by referring to the parameter name.

The output parameters will appear in the returned result set. The parameters will appear as columns in the result set in the same order as they are defined. A procedure may return one or more rows. Thus, also select statements can be wrapped into database procedures.

The following statement creates a procedure that has two input parameters and two output parameters:

```
"CREATE PROCEDURE PHONEBOOK_SEARCH
    (FIRST_NAME VARCHAR, LAST_NAME VARCHAR)
    RETURNS (PHONE_NR NUMERIC, CITY VARCHAR)
BEGIN
```

```
-- procedure_body
END";
```

This procedure should be called using two input parameter of data type VARCHAR. The procedure returns an output table consisting of 2 columns named phone_nr of type NUMERIC and CITY of type VARCHAR.

For example:

```
call phonebook_search ( 'JOHN','DOE');
Result looks like the following (when the procedure body has been
programmed)
PHONE_NR   CITY
34335556   NEW YORK
23452266   LOS ANGELES
```

Declare Section

Local variables that are used inside the procedure for temporary storage of column and control values are defined in a separate section of the stored procedure directly following the BEGIN keyword.

The syntax of declaring a variable is:

DECLARE *variable_name datatype*;

Note that every declare statement should be ended with a semicolon (;).

The variable name is an alphanumeric string that identifies the variable. The data type of the variable can be any valid SQL data type supported. For supported data types, see the appendix, "Data Types" in the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide**.

For example:

```
"CREATE PROCEDURE PHONEBOOK_SEARCH
    (FIRST_NAME VARCHAR, LAST_NAME VARCHAR)
    RETURNS (PHONE_NR NUMERIC, CITY VARCHAR)
BEGIN
DECLARE i INTEGER;

DECLARE dat DATE;
```

```
END";
```

Note that input and output parameters are treated like local variables within a procedure with the exception that input parameters have a preset value and output parameter values are returned or can be appended to the returned result set.

Procedure Body

The procedure body contains the actual stored procedure program based on assignments, expressions, SQL statements and the likes.

Any type of expression including scalar functions can be used in a procedure body. See the appendix "SOLID SQL Syntax" in the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide** for valid expressions.

Assignments

To assign values to variables either of the following syntax is used:

SET *variable_name* = *expression* ;

or

variable_name := *expression* ;

Example:

```
SET i = i+ 20 ;

i := 100;
```

Variables and constants are initialized every time a procedure is executed. By default, variables are initialized to NULL. Unless a variable has been explicitly initialized, its value is NULL, as the following example shows:

```
BEGIN
DECLARE   total   INTEGER;
...
total := total + 1;  -- assigns a null to total
...
```

Therefore, a variable should never be referenced before it has been assigned a value.

The expression following the assignment operator can be arbitrarily complex, but it must yield a data type that is the same as or convertible to the data type of the variable.

When possible, SOLID procedure language can provide conversion of data types implicitly. This makes it possible to use literals, variables and parameters of one type where another type is expected.

Implicit conversion is not possible if:

- information would be lost in the conversion.
- a string to be converted to an integer contains non-numeric data

Examples:

```
DECLARE integer_var INTEGER;
integer_var := 'NR:123';
```

returns an error.

```
DECLARE string_var CHAR(3);
string_var := 123.45;
```

results in value ‘123’ in variable *string_var*.

```
DECLARE string_var VARCHAR(2);
string_var := 123.45;
```

returns an error.

Expressions

Comparison Operators

Comparison operators compare one expression to another. The result is always TRUE, FALSE, or NULL. Typically, comparisons are used in conditional control statements and allow comparisons of arbitrarily complex expressions. The following table gives the meaning of each operator:

Operator	Meaning
=	is equal to
<>	is not equal to
<	is less than
>	is greater than

<=	is less than or equal to
>=	is greater than or equal to

Note that the **!=** notation cannot be used inside a stored procedure, use the ANSI-SQL compliant <> instead.

Logical Operators

The logical operators can be used to build more complex queries. The logical operators AND, OR, and NOT operate according to the tri-state logic illustrated by the truth tables shown below. AND and OR are binary operators; NOT is a unary operator.

NOT	**true**	**false**	**null**
	false	true	null

AND	**true**	**false**	**null**
true	true	false	null
false	false	false	false
null	null	false	null

OR	**true**	**false**	**null**
true	true	true	true
false	true	false	null
null	true	null	null

As the truth tables show, AND returns the value TRUE only if both its operands are true. On the other hand, OR returns the value TRUE if either of its operands is true. NOT returns the opposite value (logical negation) of its operand. For example, NOT TRUE returns FALSE.

NOT NULL returns NULL because nulls are indeterminate.

When not using parentheses to specify the order of evaluation, operator precedence determines the order.

Note that ‘true’ and ‘false’ are not literals accepted by SQL parser but values. Logical expression value can be interpreted as a numeric variable:

false = 0 or NULL
true = 1 or any other numeric value

Example:

`IF` *expression* `= TRUE THEN`

can be simply written

`IF` *expression* `THEN`

IS NULL Operator

The IS NULL operator returns the Boolean value TRUE if its operand is null, or FALSE if it is not null. Comparisons involving nulls always yield NULL. To test whether a value is NULL, do not use the expression,

```
IF variable = NULL THEN...
```

because it never evaluates to TRUE.

Instead, use the following statement:

```
IF variable IS NULL THEN...
```

Note that when using multiple logical operators in SOLID stored procedures the individual logical expressions should be enclosed in parentheses like:

```
((A >= B) AND (C= 2)) OR (A= 3)
```

Control Structures

IF Statement

Often, it is necessary to take alternative actions depending on circumstances. The IF statement executes a sequence of statements conditionally. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSEIF.

IF-THEN

The simplest form of IF statement associates a condition with a statement list enclosed by the keywords THEN and END IF (not ENDIF), as follows:

```
IF condition THEN
statement_list;
END IF
```

The sequence of statements is executed only if the condition evaluates to TRUE. If the condition evaluates to FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

```
IF sales > quota THEN
    SET pay = pay + bonus;
END IF
```

IF-THEN-ELSE

The second form of IF statement adds the keyword ELSE followed by an alternative statement list, as follows:

```
IF condition THEN
    statement_list1;
ELSE
    statement_list2;
END IF
```

The statement list in the ELSE clause is executed only if the condition evaluates to FALSE or NULL. Thus, the ELSE clause ensures that a statement list is executed. In the following example, the first or second assignment statement is executed when the condition is true or false, respectively:

```
IF trans_type = 'CR' THEN
    SET balance = balance + credit;
ELSE
    SET balance = balance - debit;
END IF
```

THEN and ELSE clauses can include IF statements. That is, IF statements can be nested, as the following example shows:

```
IF trans_type = 'CR' THEN
    SET balance = balance + credit ;
ELSE
    IF new_balance >= minimum_balance THEN
```

```
        SET balance = balance - debit ;
    ELSE
        SET balance = minimum_balance;
    END IF
END IF
```

IF-THEN-ELSEIF

Occasionally it is necessary to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSEIF to introduce additional conditions, as follows:

```
IF condition1 THEN
   statement_list1;
ELSEIF condition2 THEN
   statement_list2;
ELSE
   statement_list3;
END IF
```

If the first condition evaluates to FALSE or NULL, the ELSEIF clause tests another condition. An IF statement can have any number of ELSEIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition evaluates to TRUE, its associated statement list is executed and the rest of the statements (inside the IF-THEN-ELSEIF) are skipped. If all conditions evaluate to FALSE or NULL, the sequence in the ELSE clause is executed. Consider the following example:

```
IF sales > 50000 THEN
    bonus := 1500;
ELSEIF sales > 35000 THEN
    bonus := 500;
ELSE
    bonus := 100;
END IF
```

If the value of "sales" is more than 50000, the first and second conditions are true. Nevertheless, "bonus" is assigned the proper value of 1500 since the second condition is never tested. When the first condition evaluates to TRUE, its associated statement is executed and control passes to the next statement following the IF-THEN-ELSEIF.

When possible, use the ELSEIF clause instead of nested IF statements. That way, the code will be easier to read and understand. Compare the following IF statements:

```
IF condition1 THEN
    statement_list1;
ELSE
  IF condition2 THEN
     statement_list2;
  ELSE
    IF condition3 THEN
       statement_list3;
    END IF
  END IF
END IF
```

```
IF condition1 THEN
   statement_list1;
ELSEIF condition2 THEN
    statement_list2;
ELSEIF condition3 THEN
    statement_list3;
END IF
```

These statements are logically equivalent, but the first statement obscures the flow of logic, whereas the second statement reveals it.

WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
  statement_list;
END LOOP
```

Before each iteration of the loop, the condition is evaluated. If the condition evaluates to TRUE, the statement list is executed, then control resumes at the top of the loop. If the condition evaluates to FALSE or NULL, the loop is bypassed and control passes to the next statement. An example follows:

```
WHILE total <= 25000 LOOP
    ...
    total := total + salary;
END LOOP
```

The number of iterations depends on the condition and is unknown until the loop completes. Since the condition is tested at the top of the loop, the sequence might execute zero times. In the latter example, if the initial value of "total" is greater than 25000, the condition evaluates to FALSE and the loop is bypassed, altogether

Loops can be nested. When an inner loop is finished control is returned to the next loop. The procedure continues from the next statement after end loop.

Leaving Loops

It may be necessary to force the procedure to leave a loop prematurely. This can be implemented using the LEAVE keyword:

```
WHILE total < 25000 LOOP
       statement_list
       total := total + salary;
       IF exit_condition THEN
         LEAVE;
       END IF
END LOOP
statement_list2
```

Upon successful evaluation of the *exit_condition* the loop is left, and the procedure continues at the *statement list 2*.

Note

Although Solid databases support the ANSI-SQL CASE syntax, the CASE construct cannot be used inside a stored procedure as a control structure.

Handling Nulls

Nulls can cause confusing behavior. To avoid some common errors, observe the following rules:

- comparisons involving nulls always yield NULL
- applying the logical operator NOT to a null yields NULL
- in conditional control statements, if the condition evaluates to NULL, its associated sequence of statements is not executed

In the example below, you might expect the statement list to execute because "x" and "y" seem unequal. Remember though that nulls are indeterminate. Whether "x" is equal to "y" or not is unknown. Therefore, the IF condition evaluates to NULL and the statement list is bypassed.

```
x := 5;
y := NULL;
...
IF x <> y THEN  -- evaluates to NULL, not TRUE
      statement_list;  -- not executed
END IF
```

In the next example, one might expect the statement list to execute because "a" and "b" seem equal. But, again, this is unknown, so the IF condition evaluates to NULL and the statement list is bypassed.

```
a := NULL;
b := NULL;
...
IF a = b THEN  -- evaluates to NULL, not TRUE
      statement_list;  -- not executed
END IF
```

NOT Operator

Applying the logical operator NOT to a null yields NULL. Thus, the following two statements are not always equivalent:

```
IF  x > y THEN                       IF NOT x > y THEN
  high := x;                             high := y;
ELSE                                 ELSE
  high := y;                             high := x;
END IF                               END IF
```

The sequence of statements in the ELSE clause is executed when the IF condition evaluates to FALSE or NULL. If either or both "x" and "y" are NULL, the first IF statement assigns the value of "y" to "high", but the second IF statement assigns the value of "x" to "high". If neither "x" nor y" is NULL, both IF statements assign the corresponding value to "high".

Zero-Length Strings

Zero length strings are treated by a Solid server like they are a string of zero length, instead of a null. NULL values should be specifically assigned as in the following:

```
SET a = NULL;
```

This also means that checking for NULL values will return FALSE when applied to a zero-length string.

Example

Following is an example of a simple procedure that determines whether a person is an adult on the basis of a birthday as input parameter.

Note the usage of {} on scalar functions, and semicolons to end assignments and IF/END IF structures.

```
"CREATE PROCEDURE grown_up
(  birth_date DATE)
RETURNS ( description VARCHAR)
BEGIN
DECLARE temp INTEGER;
-- determine the number of years since the day of birth
temp := {fn TIMESTAMPDIFF(SQL_TSI_YEAR,birth_date,now())};
IF temp >= 18  THEN
--over 18 it's an adult
     description := 'ADULT';
ELSE
-- still a minor
     description := 'MINOR';
END IF
END";
```

Exiting a Procedure

A procedure may be exited prematurely by issuing the keyword

```
RETURN;
```

at any location. After this keyword, control is directly handed to the program calling the procedure returning the values bound to the output parameters as indicated in the returns-section of the procedure definition.

Returning Data

By default a stored procedure returns one row of data. The row is returned when the complete procedure has been run or has been forced to exit. This row conforms to the declared output parameters in the parameter section of the procedure.

It is also possible to return result sets from a procedure using the following syntax:

```
return row;
```

Every RETURN ROW call adds a new row into the returned result set.

Using SQL in a Stored Procedure

Using SQL statements inside a stored procedure is somewhat different from issuing SQL directly from tools like SOLID *DBConsole*.

Any SQL statement will have to be executed through an explicit cursor definition. A cursor is a specific allocated part of the server process memory that keeps track of the statement being processed. Memory space is allocated for holding one row of the underlying statement, together with some status information on the current row (in SELECTS) or the number of rows affected by the statement (in UPDATES, INSERTS and DELETES).

In this way query results are processed one row at a time. The stored procedure logic should take care of the actual handling of the rows, and the positioning of the cursor on the required row(s).

There are five basic steps in handling a cursor:

1. Preparing the cursor - the definition
2. Executing the cursor - executing the statement
3. Fetching on the cursor (for select procedure calls) - getting the results row by row
4. Closing the cursor after use - still enabling it to re-execute
5. Dropping the cursor from memory - definitely removing it

1. Preparing the Cursor

A cursor is defined (prepared) using the following syntax:

EXEC SQL PREPARE *cursor_name SQL_statement*;

By preparing a cursor, memory space is allocated to accommodate one row of the result set of the statement, the statement is parsed and optimized.

A cursor name given for the statement has to be unique within the connection. When a cursor is prepared a Solid server checks that no other cursor of this name is currently open. If there is one, error number 14504 is returned.

Note that statement cursors can be opened also using the ODBC API. Also these cursor names need to be different from the cursors opened from procedures.

Example:

```
EXEC SQL PREPARE sel_tables
   SELECT table_name
   FROM sys_tables
   WHERE table_name like 'SYS%';
```

This statement will prepare the cursor named *sel_tables*, but will not execute the statement that it contains.

2. Executing the Cursor

After a procedure has been successfully prepared it can be executed. An execute binds possible input and output variables to it and runs the actual statement.

Syntax of the execute statement is:

```
EXEC SQL EXECUTE cursor_name
     [ INTO  ( var1, var2, … ) ];
```

The optional section INTO binds result data of the statement to variables.

Variables listed in parenthesis after the INTO keyword are used when running a SELECT or CALL statement. The resulting columns of the SELECT or CALL statement are bound to these variables when the statement is executed. The variables are bound starting from the left-most column listed in the statement. Binding of variables continues to the following column until all variables in the list of variables have been bound. For example to extend the sequence for the cursor *sel_tables* that was prepared earlier we need to run the following statements:

```
EXEC SQL PREPARE sel_tables
   SELECT table_name
   FROM   sys_tables
   WHERE  table_name like 'SYS%'
```

```
EXEC SQL EXECUTE sel_tables INTO (tab);
```

The statement is now executed and the resulting table names will be returned into variable *tab* in the subsequent Fetch statements.

3. Fetching on the Cursor

When a SELECT or CALL statement has been prepared and executed it is ready for fetching data from it. Other statements (UPDATE,INSERT,DELETE, DDL) do not require fetching as there will be no result set. Fetching results is done using the fetch syntax:

EXEC SQL FETCH *cursor_name*;

This command fetches a single row from the cursor to the variables that were bound with INTO keyword when the statement was executed.

To complete the previous example to actually get result rows back, the statements will look like:

```
EXEC SQL PREPARE sel_tables
    SELECT table_name
    FROM   sys_tables
    WHERE  table_name like 'SYS%'
EXEC SQL EXECUTE sel_tables INTO (tab);
EXEC SQL FETCH sel_tables;
```

After this the variable *tab* will contain the table name of the first table found conforming to the WHERE-clause.

Subsequent calls to fetch on the cursor *sel_tables* will get the next row(s) if the select found more than one.

To fetch all table names a loop construct may be used:

```
WHILE expression LOOP
EXEC SQL FETCH sel_tables;
END LOOP
```

Note that after the completion of the loop the variable *tab* will contain the last fetched table name.

4. Closing the Cursor

Cursors may be closed by issuing the statement

```
EXEC SQL CLOSE cursor_name;
```

This will not remove the actual cursor definition from memory, it may be re-executed when the need arises.

5. Dropping the Cursor

Cursors may be dropped from memory, releasing all resources by the statement:

```
EXEC SQL DROP cursor_name;
```

Error Handling

SQLSUCCESS

The return value of the latest EXEC SQL statement executed inside a procedure body is stored into variable SQLSUCCESS. This variable is automatically generated for every procedure. If the previous SQL statement was successful, the value 1 is stored into SQLSUCCESS. After a failed SQL statement, a value 0 is stored into SQLSUCCESS.

The value of SQLSUCCESS may be used, for instance, to determine when the cursor has reached the end of the result set as in the following example:

```
EXEC SQL FETCH sel_tab;
-- loop as long as last statement in loop is successful
WHILE SQLSUCCESS LOOP
     -- do something with the results like return the row
   EXEC SQL FETCH sel_tab;

END LOOP
```

SQLERRNUM

This variable contains the error code of the latest SQL statement executed. It is automatically generated for every procedure. After successful execution, SQLERRNUM contains zero (0).

SQLERRSTR

This variable contains the error string from the last failed SQL statement.

SQLROWCOUNT

After the execution of UPDATE, INSERT and DELETE statements an additional variable is available to check the result of the statement. Variable SQLROWCOUNT contains the number of rows affected by the last statement.

SQLERROR

To generate user errors from procedures, the SQLERROR variable may be used to return an actual error string that caused the statement to fail to the calling application. The syntax is:

RETURN SQLERROR '*error string*'

RETURN SQLERROR *char_variable*

The error is returned in the following format:

User error: *error_string*

SQLERROR OF *cursorname*

For error checking of EXEC SQL statements the SQLSUCCESS variable may be used as described under SQLSUCCESS in the beginning of this section. To return the actual error that caused the statement to fail to the calling application, the following syntax may be used:

```
EXEC SQL PREPARE cursorname sql_statement

EXEC SQL EXECUTE cursorname

IF NOT SQLSUCCESS THEN

RETURN SQLERROR OF cursorname;

END IF
```

Processing will stop immediately when this statement is executed and the procedure return code is SQLERROR. The actual database error can be returned using the SQLError function:

```
Solid Database error 10033: Primary key unique constraint violation
```

The generic error handling method for a procedure can be declared with:

```
EXEC SQL WHENEVER SQLERROR [ROLLBACK [WORK],] ABORT;
```

When this statement is included in a stored procedure all return values of executed SQL statements are checked for errors. If a statement execution returns an error, the procedure is automatically aborted and SQLERROR of the last cursor is returned. Optionally the transaction can be rolled back.

The statement should be included before any EXEC SQL statements directly following the DECLARE section of variables.

Below is an example of a complete procedure returning all table names from SYS_TABLES that start with ‘SYS’:

```
"CREATE PROCEDURE  sys_tabs
RETURNS ( tab VARCHAR)
BEGIN
-- abort on errors
EXEC SQL WHENEVER SQLERROR ROLLBACK, ABORT;
-- prepare the cursor
EXEC SQL PREPARE sel_tables
   SELECT table_name
   FROM   sys_tables
   WHERE  table_name like 'SYS%';
-- execute the cursor
EXEC SQL EXECUTE sel_tables INTO (tab);
-- loop through rows
EXEC SQL FETCH sel_tables;
WHILE sqlsuccess LOOP
      RETURN ROW;
      EXEC SQL FETCH sel_tables;
END LOOP
-- close and drop the used cursors
EXEC SQL CLOSE sel_tables;
EXEC SQL DROP sel_tables;
END";
```

Parameter Markers in Cursors

In order to make a cursor more dynamic, a SQL statement can contain parameter markers that indicate values that are bound to the actual parameter values at execute time. The '?' symbol is used as a parameter marker.

Syntax example:

```
EXEC SQL PREPARE sel_tabs
```

```
SELECT table_name
FROM sys_tables
WHERE table_name LIKE ?
AND table_schema LIKE ?;
```

The execution statement is adapted by including a USING keyword to accommodate the binding of a variable to the parameter marker.

```
EXEC SQL EXECUTE sel_tabs USING ( var1, var2 ) INTO ( tabs);
```

In this way a single cursor can be used multiple times without having to re-prepare the cursor. As preparing a cursor involves also the parsing and optimizing of the statement, significant performance gains can be achieved by using re-usable cursors.

Note that the USING list only accepts variables, data can not be directly passed in this way. So if for example an insert into a table should be made, one column value of which should always be the same (status = ‘NEW’) then the following syntax would be wrong:

```
EXEC SQL EXECUTE ins_tab  USING (nr, desc, dat, 'NEW');
```

The correct way would be to define the constant value in the prepare section:

```
EXEC SQL PREPARE ins_tab
   INSERT INTO my_tab ( id,  descript, in_date, status)
   VALUES ( ?,?,?,'NEW');
EXEC SQL EXECUTE ins_tab USING ( nr, desc, dat);
```

Note that variables can be used multiple times in the using list.

The parameters in a SQL statement have no intrinsic data type or explicit declaration. Therefore, parameter markers can be included in a SQL statement only if their data types can be inferred from another operand in the statement.

For example, in an arithmetic expression such as ? + COLUMN1, the data type of the parameter can be inferred from the data type of the named column represented by COLUMN1. A procedure cannot use a parameter marker if the data type cannot be determined.

The following table describes how a data type is determined for several types of parameters.

Location of Parameter	Assumed Data Type
One operand of a binary arithmetic or comparison operator	Same as the other operand
The first operand in a BETWEEN clause	Same as the other operand

The second or third operand in a BETWEEN clause	Same as the first operand
An expression used with IN	Same as the first value or the result column of the subquery
A value used with IN	Same as the expression
A pattern value used with LIKE	VARCHAR
An update value used with UPDATE	Same as the update column

An application cannot place parameter markers in the following locations:

- As a SQL identifier (name of a table, name of a column etc.)
- In a SELECT list.
- As both expressions in a comparison-predicate.
- As both operands of a binary operator.
- As both the first and second operands of a BETWEEN operation.
- As both the first and third operands of a BETWEEN operation.
- As both the expression and the first value of an IN operation.
- As the operand of a unary + or - operation.
- As the argument of a set-function-reference.

For more information, see the ANSI SQL-92 specification.

In the following example, a stored procedure will read rows from one table and insert parts of them in another, using multiple cursors:

```
"CREATE PROCEDURE  tabs_in_schema (schema_nm VARCHAR)
RETURNS ( nr_of_rows INTEGER)
BEGIN
DECLARE tab_nm VARCHAR;
EXEC SQL PREPARE  sel_tab
   SELECT table_name
   FROM   sys_tables
   WHERE  table_schema = ?;
EXEC SQL PREPARE ins_tab
   INSERT INTO my_table (table_name,schema) VALUES ( ?,?);
```

```
nr_of_rows := 0;

EXEC SQL EXECUTE sel_tab USING ( schema_nm) INTO (tab_nm);
EXEC SQL FETCH sel_tab;
WHILE SQLSUCCESS LOOP
    nr_of_rows := nr_of_rows + 1;
    EXEC SQL EXECUTE ins_tab USING(tab_nm, schema_nm);
    IF SQLROWCOUNT <> 1 THEN
    RETURN SQLERROR OF ins_tab;
    END IF
    EXEC SQL FETCH sel_tab;
END LOOP
END";
```

Calling other Procedures

As calling a procedure forms a part of the supported SQL syntax, a stored procedure may be called from within another stored procedure. The default limit for levels of nested procedures is 16. When the maximum is exceeded, the transaction fails. The current nesting level is set in the `MaxNestedProcedures` parameter in the `solid.ini` configuration file. For details, see appendix, "Configuration Parameters" of the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide.**

Like all SQL statements a cursor should be prepared and executed like:

```
EXEC SQL PREPARE cp call myproc( ?,?);

EXEC SQL EXECUTE cp USING ( var1, var2);
```

If procedure *myproc* returns one or more values, then subsequently a fetch should be done on the cursor *cp* to retrieve those values:

```
EXEC SQL PREPARE cp call myproc(?,?);

EXEC SQL EXECUTE cp USING (var1, var2) INTO (ret_var1,
ret_var2);

EXEC SQL FETCH cp;
```

Note that if the called procedure uses a *return row* statement, the calling procedure should utilize a WHILE LOOP construct to fetch all results.

Recursive calls are possible, but discouraged because cursor names are unique at connection level and infinite recursion may crash the server process.

Positioned Updates and Deletes

In SOLID procedures it is possible to use positioned updates and deletes. This means that an update or delete will be done to a row where a given cursor is currently positioned. The positioned updates and deletes can also be used within stored procedures using the cursor names used within the procedure.

The following syntax is used for positioned updates:

`UPDATE` *table_name*

`SET` *column* `= value`

`WHERE CURRENT OF` *cursor_name*

and for deletes

`DELETE FROM` *table_name*

`WHERE CURRENT OF` *cursor_name*

In both cases the *cursor_name* refers to a statement doing a SELECT on the table that is to be updated/deleted from.

Positioned cursor update is a semantically suspicious concept in SQL standard that may cause peculiarities also with a Solid server. Please note the following restriction when using positioned updates.

Below is an example written with pseudo code that will cause an endless loop with a Solid server (error handling, binding variables and other important tasks omitted for brevity and clarity):

```
"CREATE PROCEDURE ENDLESS_LOOP
BEGIN
EXEC SQL PREPARE MYCURSOR SELECT * FROM TABLE1;
EXEC SQL PREPARE MYCURSOR_UPDATE UPDATE TABLE1
   SET COLUMN2 = 'new data';
EXEC SQL EXECUTE MYCURSOR;
EXEC SQL FETCH MYCURSOR;
WHILE SQLSUCCESS LOOP
   EXEC SQL EXECUTE MYCURSOR_UPDATE;
   EXEC SQL COMMIT WORK;
```

```
    EXEC SQL FETCH MYCURSOR;
END LOOP
END";
```

The endless loop is caused by the fact that when the update is committed, a new version of the row becomes visible in the cursor and it is accessed in the next FETCH statement. This happens because the incremented row version number is included in the key value and the cursor finds the changed row as the next greater key value after the current position. The row gets updated again, the key value is changed and again it will be the next row found.

In the above example, the updated column2 is not assumed to be part of the primary key for the table, and the row version number was the only index entry changed. However, if such a column value is changed that is part of the index through which the cursor has searched the data, the changed row may jump further forward or backward in the search set.

For these reasons, using positioned update is not recommended in general and searched update should be used instead whenever possible. However, sometimes the update logic may be too complex to be expressed in SQL WHERE clause and in such cases positioned update can be used as follows:

Positioned cursor update works deterministically in SOLID, when the where clause is such that the updated row does not match the criteria and therefore does not reappear in the fetch loop. Constructing such a search criteria may require using additional column only for this purpose.

Note that other users' changes do not become visible in the open cursor, only those committed within the same database session.

Transactions

Stored procedures use transactions like any other interface to the database. A transaction may be committed or rolled back either inside the procedure or outside the procedure. Inside the procedure a commit or roll back is done using the following syntax:

```
EXEC SQL COMMIT WORK;
```

```
EXEC SQL ROLLBACK WORK;
```

These statements end the previous transaction and start a new one.

If a transaction is not committed inside the procedure, it may be ended externally using:

- A SOLID API
- Another stored procedure
- By autocommit, if the connection has AUTOCOMMIT switch set to ON

Note that when a connection has autocommit activated it does not force autocommit inside a procedure. The commit is done when the procedure exits.

Default Cursor Management

By default, when a procedure exits, all cursors opened in a procedure are closed. Closing cursors means that cursors are left in a prepared state and can be re-executed.

After exiting, the procedure is put in the procedure cache. When the procedure is dropped from the cache, all cursors are finally dropped.

The number of procedures kept in cache is determined by the `solid.ini` file setting:

```
[SQL]
ProcedureCache = nbr_of_procedures
```

This means that, as long as the procedure is in the procedure cache, all cursors can be re-used as long as they are not dropped. A Solid server itself manages the procedure cache by keeping track of the cursors declared, and notices if the statement a cursor contains has been prepared.

As cursor management, especially in a heavy multi-user environment, can use a considerable amount of server resources it is good practice to always close cursors immediately and preferably also drop all cursors that are not used anymore. Only the most frequently used procedures may be left non-dropped to reduce the cursor preparation effort.

Note that transactions are not related to procedures or other statements. Commit or rollback does therefore NOT release any resources in a procedure.

Notes on SQL

- There is no restriction on the SQL statements used. Any valid SQL statement can be used inside a stored procedure, including DDL and DML statements
- Cursors may be declared anywhere in a stored procedure. Cursors that are certainly going to be used are best prepared directly following the declare section.
- Cursors that are used inside control structures, and are therefore not always necessary, are best declared at the point where they are activated, to limit the amount of open cursors and hence the memory usage.
- The cursor name is an undeclared identifier, not a variable; it is used only to reference the query. You cannot assign values to a cursor name or use it in an expression.
- Cursors may be re-executed repeatedly without having to re-prepare them. Note that this can have a serious influence on performance; repetitively preparing cursors on similar

statements may decrease the performance by around 40% in comparison to re-executing already prepared cursors!

- Any SQL statement will have to be preceded by the keywords EXEC SQL.

Functions for Procedure Stack Viewing

The following function may be included in stored procedures to analyze the current contents of the procedure stack:

- PROC_COUNT ( )

  This function returns the number of procedures in the procedure stack, including the current procedure.

- PROC_NAME (N)

  This function returns the Nth procedure name in the stack. The first procedure is in position zero.

- PROC_SCHEMA (N)

  This function returns the schema name of the Nth procedure in the procedure stack.

These functions allow for stored procedures that behave differently depending on whether they are called from an application or from a procedure.

Procedure privileges

Stored procedures are owned by the creator, and are part of the creator's schema. Users needing to run stored procedures in other schema's need to be granted EXECUTE privilege on the procedure:

`GRANT EXECUTE ON` *Proc_name* `TO USER[,ROLE];`

All database objects accessed within the granted procedure, even subsequently called procedures, are accessed according to the rights of the owner of the procedure. No special grants are necessary.

Using Triggers

A trigger activates a stored procedure code, which a Solid server automatically executes when a user attempts to change the data in a table. You may create one or more triggers on a table, with each trigger defined to activate on a specific INSERT, UPDATE, or DELETE command. When a user modifies data within the table, the trigger that corresponds to the command is activated.

Triggers enable you to:

- Implement referential integrity constraints, such as ensuring that a foreign key value matches an existing primary key value.
- Prevent users from making incorrect or inconsistent data changes by ensuring that intended modifications do not compromise a database's integrity.
- Take action based on the value of a row before or after modification.
- Transfer much of the logic processing to the backend, reducing the amount of work that your application needs to do as well as reducing network traffic.

How Triggers Work

The order in which a data manipulation statement is executed when triggers are enabled is the key to understanding how triggers work in the SOLID database.

In SOLID's DML Execution Model, a Solid server performs a number of validation checks before executing data manipulation statements (INSERT, UPDATE, or DELETE). Following is the execution order for data validation, trigger execution, and integrity constraint checking for a single DML statement.

1. Validate values if they are part of the statement (that is, not bound). This includes null value checking, data type checking (such as numeric), etc.
2. Perform table level security checks.
3. Loop for each row affected by the SQL statement. For each row perform these actions in this order:

   a. Perform column level security checks.

   b. Fire BEFORE row trigger.

   a. Validate values if they are bound in. This includes null value checks, data type checking, and size checking (for example, checking if the character string is too long).

   Note that size checking is performed even for values that are not bound.

   b. Execute INSERT/UPDATE/DELETE
   c. Fire AFTER ROW trigger
4. Commit statement
   - Perform concurrency conflict checks.
   - Perform checks for duplicate values.
   - Perform referential integrity checks on invoking DML.

Note

A trigger itself can cause the DML to be executed, which applies to the steps shown in the above model.

Creating Triggers

Use the CREATE TRIGGER (described below) to create a trigger. You can disable an existing trigger or all triggers defined on a table by using the ALTER TRIGGER commands. For details, read *"Altering Trigger Attributes"* on page 3-53. The ALTER TRIGGER command causes a Solid server to ignore the trigger when an activating DML statement is issued. With this command, you can also enable a trigger that is currently inactive.

To drop a trigger from the system catalog, use DROP TRIGGER. For details, read *"Dropping Triggers"* on page 3-52.

CREATE TRIGGER command

The CREATE TRIGGER command creates a trigger. To create a trigger you must be a DBA or owner of the table on which the trigger is being defined. To create a trigger provide the catalog, schema/owner and name of the table on which a trigger is being defined. For an example of the CREATE TRIGGER command, see *"Trigger Example"* on page 3-43.

The syntax of the CREATE TRIGGER command is:

create_trigger ::=

CREATE TRIGGER *trigger_name* ON *table_name time_of_operation*

   triggering_event [*REFERENCING column_reference*] *trigger_body*

where:

trigger_name := *literal*

time_of_operation ::= BEFORE | AFTER

triggering_event :: = INSERT | UPDATE | DELETE

column_reference ::= OLD *old_column_name* [AS] *old_col_identifier*

[*, REFERENCING column_reference* |

NEW *new_column_name* [AS] *new_col_identifier*

[*, REFERENCING column_reference*]

trigger_body ::= *trigger_body*:= [*declare_statement*] <*trigger_statement*>

{, <*trigger_statement*>]}

old_column_name := *literal*

new_column_name := *literal*

old_col_identifier := *literal*

new_col_identifier := *literal*

Keywords and Clauses

Following is a summary keywords and clauses.

Trigger_name

The *trigger_name* can contain up to 254 characters.

BEFORE | AFTER clause

The BEFORE | AFTER clause specifies whether to execute the trigger before or after the invoking DML statement, which modifies data. In some circumstances, the BEFORE and AFTER clauses are interchangeable. However, there are some situations where one clause is preferred over the other.

- It is more efficient to use the BEFORE clause when performing data validation, such as domain constraint and referential integrity checking.
- When you use the AFTER clause, table rows which become available due to the invoking DML statement are processed. Conversely, the AFTER clause also confirms data deletion after the invoking DELETE statement.

You can define up to six triggers for each combination of table, event (INSERT, UPDATE, DELETE), and time (BEFORE and AFTER). For example, you can define one trigger for each BEFORE and AFTER clause, providing 2 triggers per operation. In addition, if you provide INSERT, UPDATE, and DELETE triggers to these combinations, you have a total maximum of six triggers.

The following example shows trigger trig01 defined BEFORE INSERT ON table t1.

```
"CREATE TRIGGER TRIG01 ON T1
   BEFORE INSERT
   REFERENCING NEW COL1 AS NEW_COL1
BEGIN
   EXEC SQL PREPARE CUR1
         INSERT INTO T2 VALUES (?);
   EXEC SQL EXECUTE CUR1 USING (NEW_COL1);
END"
```

Following are examples (including implications and advantages) of using the BEFORE and AFTER clause of the CREATE TRIGGER command for each DML operation:

- UPDATE operation

  The BEFORE clause can verify that modified data follows integrity constraint rules before processing the UPDATE. If the REFERENCING NEW AS *new_column_identifier* clause is used with the BEFORE UPDATE clause, then the updated values are available to the triggered SQL statements. In the trigger, you can set the default column values or derived column values before performing an UPDATE.

  The AFTER clause can perform operations on newly modified data. For example, after a branch address update, the sales for the branch can be computed.

  If the REFERENCING OLD AS *old_column_identifier* clause is used with the AFTER UPDATE clause, then the values that existed prior to the invoking update is accessible to the triggered SQL statements.

- INSERT Operation

  The BEFORE clause can verify that modified data follows integrity constraint rules before performing an INSERT. Column values passed as parameters are visible to the triggered SQL statements but the inserted rows are not. In the trigger, you can set default column values or derived column values before performing an INSERT.

  The AFTER clause can perform operations on newly inserted data. For example, after insertion of a sales order, the total order can be computed to see if a customer is eligible for a discount.

Column values are passed as parameters and inserted rows are visible to the triggered SQL statements.

- DELETE Operation

  The BEFORE clause can perform operations on data about to be deleted. Column values passed as parameters and inserted rows are visible to the triggered SQL statements.

  The AFTER clause can be used to confirm the deletion of data. Column values passed as parameters are visible to the triggered SQL statements. Please note that the deleted rows are visible to the triggering SQL statement.

INSERT | UPDATE | DELETE Clause

The INSERT | UPDATE | DELETE clause indicates the trigger action when a user action (INSERT, UPDATE, DELETE) is attempted.

Statements related to processing a trigger occur first before commits and autocommits from the invoking DML (INSERT, UPDATE, DELETE) statements on tables. If a trigger body or a procedure called within the trigger body attempts to execute a COMMIT or ROLLBACK, than a Solid server returns an appropriate run-time error.

INSERT specifies that the trigger is activated by an INSERT on the table. Loading *n* rows of data is considered as *n* inserts.

Note

There may be some performance impact if you try to load the data with triggers enabled. Depending on your business need, you may want to disable the triggers before loading and enable them after loading. For details, see the section *"Altering Trigger Attributes"* on page 3-53.

DELETE specifies that the trigger is activated by a DELETE on the table.

UPDATE specifies that the trigger is activated by an UPDATE on the table. Note the following rules for using the UPDATE clause:

- The same column cannot be referenced by more than one UPDATE trigger.
- A Solid server allows for recursive update to the same table and does not prohibit recursive updates to the same row.

A Solid server does not detect situations where the actions of different triggers cause the same data to be updated. For example, assume there are two update triggers on different columns, Col1 and Col2, of table Tbl1. When an update is attempted on all the columns of

Tbl1, the two triggers are activated. Both triggers call stored procedures which update the same column, Col3, of a second table, Tbl2. The first trigger updates Tbl2.Col3 to 10 and the second trigger updates Tbl2.Col3 to 20.

Likewise, a Solid server does not detect situations where the result of an UPDATE which activates a trigger conflicts with the actions of the trigger itself. For example, consider the following SQL statement:

```
UPDATE t1 SET c1 = 20 WHERE c3 = 10;
```

If the trigger activated by this UPDATE then calls a procedure that contains the following SQL statement, the procedure overwrites the result of the UPDATE that activated the trigger:

```
UPDATE t1 SET c1 = 17 WHERE c1 = 20;
```

Note

The above example can lead to recursive trigger execution, which you should try to avoid.

Table_name

The *table_name* is the name of the table on which the trigger is created. Solid server allows you to drop a table that has dependent triggers defined on it. When you drop a table all dependent objects including triggers are dropped. Be aware that you may still get run-time errors. For example, assume you create two tables A and B. If a procedure SP-B inserts data into table A and the table is then dropped, a user will receive a run-time error if table B has a trigger which invokes SP-B.

Trigger_body

The *trigger_body* contains the statement(s) to be executed when a trigger fires. The *trigger_body* definition is identical to the stored procedure definition. Please *"Stored Procedures"* on page 3-1 for details on creating a trigger body.

A trigger body may also invoke any procedure registered with a Solid server. SOLID procedure invocation rules follow standard procedure invocation practices.

You must explicitly check for business logic errors and raise an error.

REFERENCING Clause

This clause is optional when creating a trigger on an INSERT/UPDATE/DELETE operation. It provides a way to reference the current column identifiers in the case of INSERT and

DELETE operations, and both the old column identifier and the new updated column identifier by aliasing the table on which an UPDATE operation occurs.

You must specify the *old_column_identifier* or the *new_col_identifier* to access them. A Solid server does not provide access to them unless you define them using the REFERENCING subclause.

OLD *old_column_name* AS *old_col_identifier* or NEW *new_column_name* AS *new_col_identifier*

This subclause of the REFERENCING clause allow you to reference the values of columns both before and after an UPDATE operation. It produces a set of old and new column values which can be passed to an inline or stored procedure; once passed, the procedure contains logic (for example, domain constraint checking) used to evaluate these parameter values.

Use the OLD AS clause to alias the table's old identifier as it exists before the UPDATE. Use the NEW AS clause to alias the table's new identifier as it exists after the UPDATE.

You cannot use the same name for the *old_column_name* and the *new_column_name*, or for the *old_column_identifier* and the *new_column_identifier.*

Each column that is referenced as NEW or OLD should have a separate REFERENCING subclause.

The statement atomicity in a trigger is such that operations made in a trigger are visible to the next SQL statements inside the trigger. For example, if you execute an INSERT statement in a trigger and then also perform a select in the same trigger, then the inserted row is visible.

In the case of AFTER trigger, an inserted row or an updated row is visible in the after insert trigger, but a deleted row cannot be seen for a select performed within the trigger. In the case of a BEFORE trigger, an inserted or updated row is invisible within the trigger and a deleted row is visible.

The table below summarizes the statement atomicity in a trigger, indicating whether the row is visible to the SELECT statement in the trigger body.

Operation	BEFORE TRIGGER	AFTER TRIGGER
INSERT	row is invisible	row is visible
UPDATE	previous value is invisible	new value is visible
DELETE	row is visible	row is invisible

Triggers Comments and Restrictions

- To use the stored procedure that a trigger calls, provide the catalog, schema/owner and name of the table on which the trigger is defined and specify whether to enable or disable the triggers in the table. For more details on stored procedures, read *"Triggers and Procedures"* on page 3-36.
- To create a trigger on a table, you must have DBA authority or be the owner of the table on which the trigger is being defined.
- You can define, by default, up to one trigger for each combination of table, event (INSERT, UPDATE, DELETE) and time (BEFORE and AFTER). This means there can be a maximum of 6 triggers per table.

Note

The triggers are applied to each row. This means that if there are 10 inserts, a trigger is executed 10 times.

- You cannot define triggers on a view (even if the view is based on a single table).
- You cannot alter a table that has a trigger defined on it when the dependent columns are affected.
- You cannot create a trigger on a system table.
- You cannot execute triggers that reference dropped or altered objects. To prevent this error:
  - Recreate any referenced object that you drop.
  - Restore any referenced object you changed back to its original state (known by the trigger).
- You can use reserved words in trigger statements if they are enclosed in double quotes. For example, the following CREATE TRIGGER statement references a column named "data" which is a reserved word.

```
"CREATE TRIGGER TRIG1 ON TMPT BEFORE INSERT
REFERENCING NEW "DATA" AS NEW_DATA
BEGIN
END"
```

Triggers and Procedures

Triggers can call stored procedures and cause a Solid server to execute other triggers. You can invoke procedures within a trigger body. In fact, you can define a trigger body, which contains procedure calls only. A procedure invoked from a trigger body can invoke other triggers.

When using stored procedures within the trigger body, you must first store the procedure with the CREATE PROCEDURE command.

In a procedure definition, you can use COMMIT and ROLLBACK statements. But in a trigger body, you *cannot* use COMMIT (including AUTOCOMMIT and COMMIT WORK) and ROLLBACK statements. You can use only the WHENEVER SQLERROR ABORT statement.

You can nest triggers up to 16 levels deep (can be changed using a configuration parameter). If a trigger gets into an infinite loop, a Solid server detects this recursive action when the 16-level nesting (or system parameter) maximum is reached and returns an error by attempting to insert an error to the user. For example, you could activate a trigger by attempting to insert into the table T1 and the trigger could call a stored procedure which also attempts to insert into T1, recursively activating the trigger.

If a set of nested triggers fails at any time, a Solid server rolls back the command which originally activated the triggers.

Setting Default or Derived Columns

You can create triggers to set up default or derived column values in INSERT and UPDATE operations. When you create the trigger for this purpose using the CREATE TRIGGER command, the trigger must follow these rules:

- The trigger must be executed BEFORE the INSERT or UPDATE operation. Column values are modified with only a BEFORE trigger. Because the column value must be set before the INSERT or UPDATE operation, using the AFTER trigger to set column values is meaningless. Note also that the DELETE operation does not apply to modifying column values.
- For an INSERT and UPDATE operation, the REFERENCING clause must contain a NEW column value for modification. Note that modifying the OLD column value is meaningless.
- New column values can be set by simply changing the variables defined in the referencing section.

Using Parameters and Variables

By using the REFERENCING clause in a trigger, old and new identifiers are captured. Variables can be passed to parameter markers used in the calling procedures or SQL statements invoked from the trigger body.

All the types of the parameters/values must be compatible with the variable types.

Triggers and Transactions

Triggers require no commit from the invoking transaction in order to fire; DML statements alone cause triggers to fire. COMMIT WORK is also disallowed in a trigger body.

In a procedure definition, you can use COMMIT and ROLLBACK statements. But in a trigger body, you *cannot* use COMMIT (including AUTOCOMMIT and COMMIT WORK) and ROLLBACK statements. You can use only the WHENEVER SQLERROR ABORT statement.

Recursion and Concurrency Conflict Errors

If a DML statement updates/deletes a row that causes a trigger to be fired, you cannot update/delete the same row again within that trigger. In such cases an AFTER trigger event can cause a recursion error and a BEFORE trigger event can cause a concurrency conflict error. For details, refer to *"Insert/Update/Delete Operations for BEFORE/AFTER Triggers"* on page 3-39.

Flawed trigger logic occurs in the following example in which the same row is deleted in a BEFORE UPDATE trigger; this causes SOLID to generate a concurrency conflict error.

```
DROP EMP;
COMMIT WORK;

CREATE TABLE EMP(C1 INTEGER);
INSERT INTO EMP VALUES (1);
COMMIT WORK;

"CREATE TRIGGER TRIG1 ON EMP
   BEFORE UPDATE
   REFERENCING OLD C1 AS OLD_C1
BEGIN
   EXEC SQL WHENEVER SQLERROR ABORT;
   EXEC SQL CUR1 DELETE FROM EMP WHERE C1 = ?;
```

```
    EXEC SQL EXECUTE CUR1 USING (OLD_C1);
    END";

UPDATE EMP SET C1=200 WHERE C1 = 1;
SELECT * FROM EMP;

ROLLBACK WORK;
```

Note

If the row that is updated/deleted were based on a unique key, instead of an ordinary column (as in the example above), SOLID generates the following error message: **1001: key value not found.**

To avoid recursion and concurrency conflict errors, be sure to check the application logic and take precautions to insure the application does not cause two transactions to update or delete the same row.

In the following table, trigger actions for insert/update/delete operations for BEFORE and AFTER triggers are detailed below. The table shows the expected results of the trigger action for the lock type used.

Insert/Update/Delete Operations for BEFORE/AFTER Triggers

Trigger	Operation	Trigger Action	Lock Type	Result
AFTER	INSERT	UPDATE the same row by adding a number to the value	Optimistic	Record is updated.
AFTER	INSERT	UPDATE the same row by adding a number to the value	Pessimistic	Record is updated.
BEFORE	INSERT	UPDATE the same row by adding a number to the value	Optimistic	Record is not updated since the WHERE condition of the UPDATE within the trigger body returns a NULL result-set (as the desired row is not yet inserted in the table).
BEFORE	INSERT	UPDATE the same row by adding a number to the value	Pessimistic	Record is not updated since the WHERE condition of the UPDATE within the trigger body returns a NULL result-set (as the desired row is not yet inserted in the table).
AFTER	INSERT	DELETE the same row that is being inserted	Optimistic	Record is deleted.
AFTER	INSERT	DELETE the same row that is being inserted	Pessimistic	Record is deleted.
BEFORE	INSERT	DELETE the same row that is being inserted	Optimistic	Record is not deleted since the WHERE condition of the DELETE within the trigger body returns a NULL result-set (as the desired row is not yet inserted in the table).
BEFORE	INSERT	DELETE the same row that is being inserted	Pessimistic	Record is not updated since the WHERE condition of the UPDATE within the trigger body returns a NULL result-set (as the desired row is not yet inserted in the table).
AFTER	UPDATE	UPDATE the same row by adding a number to the value	Optimistic	Generates SOLID Table Error: Too many nested triggers.
AFTER	UPDATE	UPDATE the same row by adding a number to the value	Pessimistic	Generates SOLID Table Error: Too many nested triggers.
BEFORE	UPDATE	UPDATE the same row by adding a number to the value	Optimistic	Record is updated, but does not get into a nested loop because the WHERE condition in the trigger body returns a NULL resultset and no rows are updated to fire the trigger recursively.

Trigger	Operation	Trigger Action	Lock Type	Result
BEFORE	UPDATE	UPDATE the same row by adding a number to the value.	Pessimistic	Record is updated, but does not get into a nested loop because the WHERE condition in the trigger body returns a NULL resultset and no rows are updated to fire the trigger recursively.
AFTER	UPDATE	DELETE the same row that is being inserted	Optimistic	Record is deleted.
AFTER	UPDATE	DELETE the same row that is being inserted	Pessimistic	Record is deleted.
BEFORE	UPDATE	DELETE the same row that is being inserted.	Optimistic	Record is updated.
BEFORE	UPDATE	DELETE the same row that is being inserted.	Pessimistic	Record is updated.
AFTER	DELETE	INSERT a row with the same value.	Optimistic	Same record is inserted after deleting.
AFTER	DELETE	INSERT a row with the same value.	Pessimistic	Hangs at the time of firing the trigger.
BEFORE	DELETE	INSERT a row with the same value.	Optimistic	Same record is inserted after deleting
BEFORE	DELETE	INSERT a row with the same value.	Pessimistic	Hangs at the time of firing the trigger.
AFTER	DELETE	INSERT a row with the same value.	Optimistic	Record is deleted.
AFTER	DELETE	UPDATE the same row by adding a number to the value.	Pessimistic	Record is deleted.
BEFORE	DELETE	UPDATE the same row by adding a number to the value.	Optimistic	Record is deleted.
BEFORE	DELETE	UPDATE the same row by adding a number to the value	Pessimistic	Record is deleted.

Error Handling

If a procedure returns an error to a trigger, the trigger causes its invoking DML command to fail with an error. To automatically return errors during the execution of a DML statement, you must use WHENEVER SQLERROR ABORT statement in the trigger body. Otherwise, errors must be checked explicitly within the trigger body after each procedure call or SQL statement.

For any errors in the user written business logic as part of the trigger body, users must use the RETURN SQLERROR statement. For details, see *"Trigger Execution Errors"* on page 3-43.

If RETURN SQLERROR is not specified, then the system returns a default error message when the SQL statement execution fails. Any changes to the database due to the current DML statement are undone and the transaction is still active. In effect, transactions are not rolled back if a trigger execution fails, but the current executing statement is rolled back.

Note

Triggered SQL statements are a part of the invoking transaction. If the invoking DML statement fails due to either the trigger or another error that is generated outside the trigger, all SQL statements within the trigger are rolled back along with the failed invoking DML command.

It is the responsibility of the invoking transaction to commit or rollback any DML statements executed within the trigger's procedure. However, this rule does not apply if the DML command invoking the trigger fails as a result of the associated trigger. In this case, any DML statements executed within that trigger's procedure are automatically rolled back.

The COMMIT and ROLLBACK statements must be executed outside the trigger body and cannot be executed within the trigger body. If one executes COMMIT or ROLLBACK within the trigger body or within a procedure called from the trigger body or another trigger, the user will get a run-time error.

Nested and Recursive Triggers

If a trigger gets into an infinite loop, a Solid server detects this recursive action when the 16-level nesting (or `MaxNestedTriggers` system parameter maximum is reached). For example, an insert attempt on table T1 activates a trigger and the trigger could call a stored procedure which also attempts to insert into Table T1, recursively activating the trigger. A Solid server returns an error on a user's insert attempt.

If a set of nested triggers fails at any time, a Solid server rolls back the command which originally activated the triggers.

Triggers and Referential Integrity

A Solid server supports referential integrity constraints. However, triggers are useful for implementing referential integrity constraints that are not supported by standard declarative referential integrity provided by a Solid server. For example, you can use triggers to implement an UPDATE CASCADE or UPDATE SET NULL constraint.

You may also use triggers to implement DELETE constraints. A Solid server does not support DELETE constraints. For example, you can specify trigger logic for each parent/dependent relationship. When a row is deleted from a parent table, you can delete all dependent child records using the associated trigger body.

Note that when using triggers to enforce referential integrity rules (instead of Solid server's declarative referential integrity) no cycle or conflict checks are performed.

Referential integrity checks on the invoking DML statement are always made after a BEFORE trigger is fired but before an AFTER trigger is fired.

Trigger Privileges and Security

Because triggers can be activated by a user's attempt to INSERT, UPDATE, or DELETE data, no privileges are required to execute them.

When a user invokes a trigger, the user assumes the privileges of the owner of the table on which the trigger is defined. The action statements are executed on behalf of the table owner, not the user who activates the trigger. However, to create a trigger which uses a stored procedure requires that the creator of the trigger meet one of the following conditions:

- You have DBA privileges.
- You are the owner of the table on which the trigger is being defined.
- You were granted all privileges on the table.

If the creator has DBA authority and creates a table for another user, a Solid server assumes that unqualified names specified in the TRIGGER command belong to the user. For example, the following command is executed under DBA authority:

```
CREATE TRIGGER A.TRIG ON EMP BEFORE UPDATE
```

Since the EMP table is unqualified, the Solid server assumes that the qualified table name is A.EMP, not DBA.EMP.

Trigger Execution Errors

At times, it is possible to receive an error in executing a trigger. The error may be due to execution of SQL statements or business logic.

Users can receive any errors in a procedure variable using the SQL statement:

RETURN SQLERROR *error_string*

or

RETURN SQLERROR *char_variable*

The error is returned in the following format:

User error: *error_string*

If a user does not specify the RETURN SQLERROR statement in the trigger body, then all trapped SQL errors are raised with a default *error_string* determined by the system. For details, see the appendix, "Error Codes" in the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide**.

Trigger Example

```
DROP TABLE TRIGGER_TEST;
DROP TABLE TRIGGER_ERR_TEST;
DROP TABLE TRIGGER_ERR_B_TEST;
DROP TABLE TRIGGER_ERR_A_TEST;
DROP TABLE TRIGGER_OUTPUT;
COMMIT WORK;

CREATE TABLE TRIGGER_TEST(
        XX VARCHAR,
        BI VARCHAR,
        AI VARCHAR,
```

```
        BU VARCHAR,
        AU VARCHAR,
        BD VARCHAR,
        AD VARCHAR
);
COMMIT WORK;

-- Table for 'before' trigger errors
CREATE TABLE TRIGGER_ERR_B_TEST(
        XX VARCHAR,
        BI VARCHAR,
        AI VARCHAR,
        BU VARCHAR,
        AU VARCHAR,
        BD VARCHAR,
        AD VARCHAR
);
INSERT INTO TRIGGER_ERR_B_TEST VALUES('x','x','x','x','x',
     'x','x');
COMMIT WORK;

-- Table for 'after X' trigger errors
CREATE TABLE TRIGGER_ERR_A_TEST(
        XX VARCHAR,
        BI VARCHAR,
        AI VARCHAR,
        BU VARCHAR,
        AU VARCHAR,
        BD VARCHAR,
        AD VARCHAR
);
INSERT INTO TRIGGER_ERR_A_TEST VALUES('x','x','x','x','x',
      'x','x');
```

```
COMMIT WORK;
 CREATE TABLE TRIGGER_OUTPUT(
        TEXT VARCHAR,
        NAME VARCHAR,
        SCHEMA VARCHAR
);
COMMIT WORK;
 ----------------------------------------------------------------
 Success triggers
----------------------------------------------------------------
 "CREATE TRIGGER TRIGGER_BI ON TRIGGER_TEST
        BEFORE INSERT
        REFERENCING NEW BI AS NEW_BI
BEGIN
        EXEC SQL PREPARE BI INSERT INTO TRIGGER_OUTPUT VALUES(
           'BI', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE BI;
        SET NEW_BI = 'TRIGGER_BI';
END";
COMMIT WORK;
 "CREATE TRIGGER TRIGGER_AI ON TRIGGER_TEST
        AFTER INSERT
        REFERENCING NEW AI AS NEW_AI
BEGIN
        EXEC SQL PREPARE AI INSERT INTO TRIGGER_OUTPUT VALUES(
           'AI', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE AI;
        SET NEW_AI = 'TRIGGER_AI';
END";
COMMIT WORK;
 "CREATE TRIGGER TRIGGER_BU ON TRIGGER_TEST
        BEFORE UPDATE
        REFERENCING NEW BU AS NEW_BU
```

```
BEGIN
        EXEC SQL PREPARE BU INSERT INTO TRIGGER_OUTPUT VALUES(
           'BU', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE BU;
        SET NEW_BU = 'TRIGGER_BU';
END";
COMMIT WORK;
 "CREATE TRIGGER TRIGGER_AU ON TRIGGER_TEST
        AFTER UPDATE
        REFERENCING NEW AU AS NEW_AU
BEGIN
        EXEC SQL PREPARE AU INSERT INTO TRIGGER_OUTPUT VALUES(
           'AU', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE AU;
        SET NEW_AU = 'TRIGGER_AU';
END";
COMMIT WORK;
 "CREATE TRIGGER TRIGGER_BD ON TRIGGER_TEST
        BEFORE DELETE
        REFERENCING OLD BD AS OLD_BD
BEGIN
        EXEC SQL PREPARE BD INSERT INTO TRIGGER_OUTPUT VALUES(
           'BD', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE BD;
        SET OLD_BD = 'TRIGGER_BD';
END";
COMMIT WORK;

"CREATE TRIGGER TRIGGER_AD ON TRIGGER_TEST
        AFTER DELETE
        REFERENCING OLD AD AS OLD_AD
BEGIN
        EXEC SQL PREPARE AD INSERT INTO TRIGGER_OUTPUT VALUES(
```

```
          'AD', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE AD;
        SET OLD_AD = 'TRIGGER_AD';
END";
COMMIT WORK;

-----------------------------------------------------------------
Error in trigger create, wrong error variable type.
-----------------------------------------------------------------

"CREATE TRIGGER TRIGGER_ERR_AU ON TRIGGER_ERR_A_TEST
        AFTER UPDATE
        REFERENCING NEW AU AS NEW_AU
BEGIN
        DECLARE ERRSTR INTEGER;
        EXEC SQL PREPARE AU INSERT INTO TRIGGER_OUTPUT VALUES(
          'AU', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE AU;
        SET NEW_AU = 'TRIGGER_AU';
        RETURN SQLERROR ERRSTR;
END";
COMMIT WORK;
-----------------------------------------------------------------
Error triggers
-----------------------------------------------------------------
"CREATE TRIGGER TRIGGER_ERR_BI ON TRIGGER_ERR_B_TEST
        BEFORE INSERT
        REFERENCING NEW BI AS NEW_BI
BEGIN
        EXEC SQL PREPARE BI INSERT INTO TRIGGER_OUTPUT VALUES(
          'BI', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE BI;
        SET NEW_BI = 'TRIGGER_BI';
```

```
        RETURN SQLERROR 'Error in TRIGGER_ERR_BI';
END";
COMMIT WORK;
"CREATE TRIGGER TRIGGER_ERR_AI ON TRIGGER_ERR_A_TEST
        AFTER INSERT
        REFERENCING NEW AI AS NEW_AI
BEGIN
        EXEC SQL PREPARE AI INSERT INTO TRIGGER_OUTPUT VALUES(
          'AI', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE AI;
        SET NEW_AI = 'TRIGGER_AI';
        RETURN SQLERROR 'Error in TRIGGER_ERR_AI';
END";
COMMIT WORK;

"CREATE TRIGGER TRIGGER_ERR_BU ON TRIGGER_ERR_B_TEST
        BEFORE UPDATE
        REFERENCING NEW BU AS NEW_BU
BEGIN
        EXEC SQL PREPARE BU INSERT INTO TRIGGER_OUTPUT VALUES(
          'BU', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE BU;
        SET NEW_BU = 'TRIGGER_BU';
        RETURN SQLERROR 'Error in TRIGGER_ERR_BU';
END";
COMMIT WORK;

"CREATE TRIGGER TRIGGER_ERR_AU ON TRIGGER_ERR_A_TEST
        AFTER UPDATE
        REFERENCING NEW AU AS NEW_AU
BEGIN
        DECLARE ERRSTR VARCHAR;
        EXEC SQL PREPARE AU INSERT INTO TRIGGER_OUTPUT VALUES(
```

```
            'AU', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE AU;
        SET NEW_AU = 'TRIGGER_AU';
        SET ERRSTR = 'Error in TRIGGER_ERR_AU';
        RETURN SQLERROR ERRSTR;
END";
COMMIT WORK;

"CREATE TRIGGER TRIGGER_ERR_BD ON TRIGGER_ERR_B_TEST
        BEFORE DELETE
        REFERENCING OLD BD AS OLD_BD
BEGIN
        EXEC SQL PREPARE BD INSERT INTO TRIGGER_OUTPUT VALUES(
            'BD', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE BD;
        SET OLD_BD = 'TRIGGER_BD';
        RETURN SQLERROR 'Error in TRIGGER_ERR_BD';
END";
COMMIT WORK;

"CREATE TRIGGER TRIGGER_ERR_AD ON TRIGGER_ERR_A_TEST
        AFTER DELETE
        REFERENCING OLD AD AS OLD_AD
BEGIN
        EXEC SQL PREPARE AD INSERT INTO TRIGGER_OUTPUT VALUES(
            'AD', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE AD;
        SET OLD_AD = 'TRIGGER_AD';
        RETURN SQLERROR 'Error in TRIGGER_ERR_AD';
END";
COMMIT WORK;

------------------------------------------------------------------
```

```
Success trigger tests
-----------------------------------------------------------------

INSERT INTO TRIGGER_TEST(XX) VALUES ('XX');
COMMIT WORK;

SELECT * FROM TRIGGER_TEST;
COMMIT WORK;

UPDATE TRIGGER_TEST SET XX = 'XX updated';
COMMIT WORK;

SELECT * FROM TRIGGER_TEST;
COMMIT WORK;

DELETE FROM TRIGGER_TEST;
COMMIT WORK;

SELECT * FROM TRIGGER_TEST;
SELECT * FROM TRIGGER_OUTPUT;
COMMIT WORK;

-----------------------------------------------------------------
Error trigger tests
-----------------------------------------------------------------

INSERT INTO TRIGGER_ERR_B_TEST(XX) VALUES ('XX');
COMMIT WORK;

SELECT * FROM TRIGGER_ERR_B_TEST;
COMMIT WORK;

UPDATE TRIGGER_ERR_B_TEST SET XX = 'XX updated';
```

```
COMMIT WORK;

SELECT * FROM TRIGGER_ERR_B_TEST;
COMMIT WORK;

DELETE FROM TRIGGER_ERR_B_TEST;
COMMIT WORK;

SELECT * FROM TRIGGER_ERR_B_TEST;
SELECT * FROM TRIGGER_OUTPUT;
COMMIT WORK;

INSERT INTO TRIGGER_ERR_A_TEST(XX) VALUES ('XX');
COMMIT WORK;

SELECT * FROM TRIGGER_ERR_A_TEST;
COMMIT WORK;

UPDATE TRIGGER_ERR_A_TEST SET XX = 'XX updated';
COMMIT WORK;

SELECT * FROM TRIGGER_ERR_A_TEST;
COMMIT WORK;

DELETE FROM TRIGGER_ERR_A_TEST;
COMMIT WORK;

SELECT * FROM TRIGGER_ERR_A_TEST;
SELECT * FROM TRIGGER_OUTPUT;
COMMIT WORK;
```

Dropping Triggers

To drop a trigger defined on a table, use the DROP TRIGGER command. This command drops the trigger from the system catalog.

You must be the owner of a table, or a user with DBA authority to drop a trigger from the table.

The syntax is:

DROP TRIGGER [*catalog_name*[*schema_name*]]*trigger_name*

DROP TRIGGER *trigger_name*
DROP TRIGGER *schema_name.trigger_name*
DROP TRIGGER c*atalog_name.schema_name.trigger_name*

The *trigger_name* is name of the trigger on which the table is defined.

If the trigger is part of a schema, indicate the schema name as in:

schema_name.*trigger_name*

If the trigger is part of a catalog, indicate the catalog name as in:

catalog_name.*schema_name*.*trigger_name*

Example of Dropping and Recreating a Trigger

```
DROP TRIGGER TRIGGER_BI;
COMMIT WORK;

"CREATE TRIGGER TRIGGER_BI ON TRIGGER_TEST
        BEFORE INSERT
        REFERENCING NEW BI AS NEW_BI
BEGIN
        EXEC SQL PREPARE BI INSERT INTO TRIGGER_OUTPUT VALUES(
          'BI_NEW', TRIG_NAME(0), TRIG_SCHEMA(0));
        EXEC SQL EXECUTE BI;
        SET NEW_BI = 'TRIGGER_BI_NEW';
END";
COMMIT WORK;

INSERT INTO TRIGGER_TEST(XX) VALUES ('XX');
```

```
COMMIT WORK;

SELECT * FROM TRIGGER_TEST;
SELECT * FROM TRIGGER_OUTPUT;
COMMIT WORK;
```

Altering Trigger Attributes

You can alter trigger attributes using the ALTER TRIGGER command. The valid attributes are ENABLED and DISABLED trigger.

The ALTER TRIGGER command causes a Solid server to ignore the trigger when an activating DML statement is issued. With this command, you can also enable a trigger that is currently inactive or disable a trigger that is currently defined on a table.

You must be the owner of a table, or a user with DBA authority to alter a trigger from the table.

alter_trigger :=

ALTER TRIGGER *trigger_name_att* SET ENABLED | DISABLED

trigger_name_attr := [*catalog_name*.[*schema_name*]]*trigger_name*

Example

```
ALTER TRIGGER SET ENABLED trig_on_employee;
```

Obtaining Trigger Information

You obtain trigger information by using trigger functions that return specific information and performing a query on the trigger system table. Each of these sources is described in this section.

Trigger Functions

The following system supported triggers stack functions are useful for analyzing and debugging purposes.

Note

The trigger stack refer to those triggers that are cached, regardless of whether they are executed or detected for execution. Trigger stack functions can be used in the application program like any other function.

The functions are:

- TRIG_COUNT ()

  This function returns the number of triggers in the trigger stack, including the current trigger. The return value is an integer.

- TRIG_NAME (n)

  This function returns the nth trigger name in the trigger stack. The first trigger position or offset is zero.

- TRIG_SCHEMA (n)

  This function returns the nth trigger schema name in the trigger stack. The first trigger position or offset is zero. The return value is a string.

Trigger System Table

Triggers are stored in a system table called SYS_TRIGGERS. The following is the meta data for the SYS_TRIGGERS system table:

Column name	**Data type**	**Description**
ID	INTEGER	unique table identifier
TRIGGER_NAME	WVARCHAR	trigger name
TRIGGER_TEXT	LONG WVARCHAR	trigger body
TRIGGER_BIN	LONG VARBINARY	compiled form of the trigger
TRIGGER_SCHEMA	WVARCHAR	the owner
CREATIME	TIMESTAMP	the creation time of the trigger
TYPE	INTEGER	reserved for future use
REL_ID	INTEGER	the relation id
PRIMARY KEY (ID) UNIQUE (TRIGGER_NAME, TRIGGER_SCHEMA) UNIQUE (REL_ID, TYPE)		

Trigger Parameter Settings

Setting Nested Trigger Maximum

Triggers can invoke other triggers or a trigger can invoke itself (or recursive triggers). The maximum number of nested or recursive triggers can be configured by the `MaxNestedTriggers` system parameter in the SQL section of SOLID.INI.

```
[SQL] MaxNestedTriggers = n;
```

where n is the maximum number of nested triggers.

The default number for nested triggers is 16.

Setting the Trigger Cache

In a Solid server, triggers are cached in a separate cache. Each user has a separate cache for triggers. As the triggers are executed, the trigger procedure logic is cached in the trigger cache and is reused when the trigger is executed again.

You can set the size of the trigger cache using the `TriggerCache` system parameter in the SQL section of SOLID.INI.

```
[SQL] TriggerCache = n;
```

where n is the number of triggers being reserved for the cache.

Using Sequences

A sequence object is used to get sequence numbers. The syntax is:

`CREATE [DENSE] SEQUENCE` *sequence_name*

Depending on how the sequence is created, there may or may not be holes in the sequence (the sequence can be sparse or dense). Dense sequences guarantee that there are no holes in the sequence numbers. The sequence number allocation is bound to the current transaction. If the transaction rolls back, also the sequence number allocations are rolled back. The drawback of dense sequences is that the sequence is locked out from other transactions until the current transaction ends.

If there is no need for dense sequences, a sparse sequence can be used. A sparse sequence guarantees uniqueness of the returned values, but it is not bound to the current transaction. If a transaction allocates a sparse sequence number and later rolls back, the sequence number is simply lost.

A sequence object can be used, for example, to generate primary key numbers. The advantage of using a sequence object instead of a separate table is that the sequence object is specifically fine-tuned for fast execution and requires less overhead than normal update statements.

Both dense and sparse sequence numbers start from 1.

After creating the sequence with the CREATE SEQUENCE statement, you can access the Sequence object values by using the following constructs in SQL statements:

- *sequencename*.CURRVAL which returns the current value of the sequence
- *sequencename*.NEXTVAL which increments the sequence by one and returns the next value.

An example of creating unique identifiers automatically for a table is given below:

```
INSERT INTO ORDERS (id, ...)
VALUES (order_seq.NEXTVAL, ...);
```

Sequences can also be used inside stored procedures. The current sequence value can be retrieved using the following statement:

`EXEC SEQUENCE` *sequence_name*`.CURRENT INTO` *variable*`;`

New sequence values can be retrieved using the following syntax:

`EXEC SEQUENCE` *sequence_name*`.NEXT INTO` *variable*`;`

It is also possible to set the current value of a sequence to a predefined value by using the following syntax:

`EXEC SEQUENCE` *sequence_name* `SET VALUE USING` *variable*`;`

An example of using a stored procedure to retrieve a new sequence number is given below:

```
"CREATE PROCEDURE get_my_seq
RETURNS (val INTEGER)
BEGIN
EXEC SEQUENCE my_sequence.NEXT INTO (val);
END";
```

Using Events

Event alerts are special objects in a SOLID database. They are used for sending events from one application to another. The use of event alerts removes resource consuming database polling from applications.

The system does not automatically generate events, they must be triggered by stored procedures. Similarly the events can only be received in stored procedures. When an application calls a stored procedure that waits for a specific event to happen, the application is blocked until the event is triggered and received. In multithreaded environments separate threads and connections can be used to access the database during the event standstill.

An event has a name that identifies it and a set of parameters. The name can be any user-specified alphanumeric string. An event object is created with the SQL statement:

`CREATE EVENT` *event_name*

`[(`*parameter_name datatype*

`[`*parameter_name datatype*`...])]`

The parameter list specifies parameter names and parameter types. The parameter types are normal SQL types. Events are dropped with the SQL statement:

`DROP EVENT` *event_name*

Events are triggered and received inside stored procedures. Special stored procedure statements are used to trigger and receive events.

The event is triggered with the stored procedure statement

`POST EVENT` *event_name* `(`*parameters*`)`

Event parameters must be local variables or parameters in the stored procedure where the event is triggered. All clients that are waiting for the posted event will receive the event.

To make a procedure wait for an event to happen, the WAIT EVENT construct is used in the stored procedure:

wait_event_statement`::=`

`WAIT EVENT`

`[`*event_specification*`...]`

`END WAIT`

event_specification`::=`

`WHEN` *event_name* `(`*parameters*`) BEGIN`

statements

`END EVENT`

Event Example

Example of a procedure that waits for an event:

```
"create procedure event_wait(i1 integer)
returns (result varchar)
begin
declare i integer;
declare c char(4);

i := 0;


wait event
   when test1 begin
      result := 'event1';
      return;
   end event


   when test2(i) begin
   end event


   when test3(i, c) begin
  end event
end wait


if i <> 0 then
   result := 'if';
   post event test1;
else
   result := 'else';
```

```
    post event test2(i);
    post event test3(i, c);
end if
end";
```

4

Using UNICODE

This chapter describes how to implement the UNICODE standard, providing the capability to encode characters used in the major languages of the world. Topics in this chapter include:

- What is UNICODE?
- UNICODE and SOLID databases
- Setting up a SOLID database for UNICODE data
- Using UNICODE with SOLID *ODBC Driver*
- Using UNICODE with the SOLID *JDBC Driver*

What is Unicode?

The Unicode Standard is the universal character encoding standard used for representation of text for computer processing. Unicode provides a consistent way of encoding multilingual plain text making it easier to exchange text files internationally.

The version 2.0 Unicode Standard is fully compatible with the International Standard ISO/IEC 10646-1; 1993, and contains all the same characters and encoding points as ISO/IEC 10646. This code-for-code identity is true for all encoded characters in the two standards, including the East Asian (Han) ideographic characters. The Unicode Standard also provides additional information about the characters and their use. Any implementation that conforms to Unicode also conforms to ISO/IEC 10646.

Unicode uses a 16-bit encoding that provides code points for more than 65,000 characters. To keep character coding simple and efficient, the Unicode Standard assigns each character a unique 16-bit value, and does not use complex modes or escape codes.

While 65,000 characters are sufficient for encoding most of the many thousands of characters used in major languages of the world, the Unicode standard and ISO 10646 provide an extension mechanism called UTF-16 that allows for encoding as many as a million more characters, without use of escape codes. This is sufficient for all known character encoding requirements, including full coverage of all historic scripts of the world.

What Characters Does the Unicode Standard Include?

The Unicode Standard defines codes for characters used in the major languages written today. This includes punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, dingbats, etc. In all, the Unicode Standard provides codes for nearly 39,000 characters from the world's alphabets, ideograph sets, and symbol collections.

There are about 18,000 unused code values for future expansion in the basic 16-bit encoding, plus provision for another 917,504 code values through the UTF-16 extension mechanism. The Unicode Standard also reserves 6,400 code values for private use, which software and hardware developers can assign internally for their own characters and symbols. UTF-16 makes another 131,072 private use code values available, should 6,400 be insufficient for particular applications.

Encoding Forms

Character encoding standards define not only the identity of each character and its numeric value, or code position, but also how this value is represented in bits. The Unicode Standard endorses two forms that correspond to ISO 10646 transformation formats, UTF-8 and UTF-16.

The ISO/IEC 10646 transformation formats UTF-8 and UTF-16 are essentially ways of turning the encoding into the actual bits that are used in implementation. The first is known as UTF-16. It assumes 16-bit characters and allows for a certain range of characters to be used as an extension mechanism in order to access an additional million characters using 16-bit character pairs. The Unicode Standard, Version 2.0, has adopted this transformation format as defined in ISO/IEC 10646.

The other transformation format is known as UTF-8. This is a way of transforming all Unicode characters into a variable length encoding of bytes. It has the advantages that the Unicode characters corresponding to the familiar ASCII set end up having the same byte values as ASCII, and that Unicode characters transformed into UTF-8 can be used with much existing software without extensive software rewrites. The Unicode Consortium also endorses the use of UTF-8 as a way of implementing the Unicode Standard. Any Unicode character expressed in the 16-bit UTF-16 form can be converted to the UTF-8 form and back without loss of information.

The international standard ISO/IEC 10646 allows for two forms of use, a two-octet (=byte) form known as UCS-2 and a four-octet form known as UCS-4. The Unicode Standard, as a profile of ISO/IEC 10646, chooses the two-octet form, which is equivalent character representationin 16-bits per character. When extended characters are used, Unicode is equivalent to UTF-16.

Implementing Unicode

This section contains pertinent information required to implement the Unicode standard in SOLID *Embedded Engine* 3.5 and SOLID *SynchroNet* 2.0. Please note the following implementation guidelines:

- Unicode data types

  SQL data types WCHAR, WVARCHAR and LONG WVARCHAR are used to store Unicode data in a Solid database. The “Wide-character” implementation conforms to ODBC 3.5 specification. The Unicode data types are interoperable with corresponding character data types (CHAR, VARCHAR and LONG VARCHAR), but conversions from Unicode data types to character data types fail, if the characters are beyond ISO Latin 1. All string operations are possible between Unicode and character data types with implicit type conversions.

- I**nternal storage format**

  The storage format (in SOLID *Embedded Engine* 3.5 and SOLID *SynchroNet* 2.0) for Unicode column data is UCS-2. All character information in the data dictionary are stored as Unicode. To support Unicode you must convert all databases created prior to the release of SOLID *Embedded Engine* version 3.x and SOLID *SynchroNet* 1.1 to support Unicode. For details, please refer to the latest release notes.

  The wide character types require more storage space than normal character types. Therefore, use wide characters only where necessary.

- Ordering data columns

  Unicode data columns are ordered based on the binary values of the UCS-2 format. If the binary order is different than what natural language users expect, developers need to provide a separate column to store the correct ordering information.

- Unicode File Names

  A Solid server does not support using Unicode strings in any file names.

Setting Up Unicode Data

Creating Columns for Storing Unicode Data

In order to start storing Unicode data in a SOLID database, tables with Unicode data columns need to be created first as follows:

```
CREATE TABLE customer (c_id INTEGER, c_name WVARCHAR,…)
```

Loading Unicode Data

You can use the data import tool *Speedloader* from SOLID version 3.5 to import data to Unicode columns. The import files should contain Unicode data in UTF-8 format.

Using Unicode in Database Entity Names

It is possible to name tables, columns, procedures, etc. with Unicode strings, simply by enclosing the Unicode names with double quotes in all the SQL statements.

The SOLID tools, like *DBConsole*, will handle Unicode strings in UTF-8 format. In order to enter native Unicode strings, third-party database administration applications need to be used, or a special application using SOLID *JDBC Driver* 2.0 should be written for this purpose.

Unicode User Names and Passwords

User names and passwords can also be Unicode strings. However, to avoid access problems from different tools, the original database administrator account information must be given as pure ASCII strings.

SOLID *Data Dictionary*, SOLID *Export,* and SOLID *Speedloader*

The SOLID Tools use UTF-8 as the external representation format of Unicode strings.

SOLID *Speedloader* (`solload`) accepts Unicode data in control and input files in UTF-8 format.

SOLID *Export* (`solexp`) extracts Unicode data from database to output files in UTF-8 format.

SOLID *Data Dictionary* (`soldd`) prints table, column, etc. names containing Unicode strings in UTF-8 format into the SQL DDL file.

Note that the teletype SOLID *SQL Editor* (`solsql`) can use the SQL files output by `soldd` to create the tables, indices, etc. for a new database, as well as data definition entries if Unicode strings are available for them.

SOLID *Data Dictionary* and SOLID *Export* accept option -8 to allow exporting data dictionary information in 8-bit format for use with SOLID *Embedded Engine (*formerly SOLID *Server)* 2.x tools. The option -8 is needed if there are scandinavian or other national non-ascii characters in the data dictionary names.

SOLID *DBConsole and teletype tools*

SOLID *DBConsole*, which requires Java 2.0, JDK 1.2, and the JDBC 2.0 driver, supports Unicode data. The teletype versions of SOLID *SQL Editor* and *Remote Control*, `solsql` and `solcon`, will function correctly in Unicode client environments.

UNICODE and SOLID *ODBC Driver*

The SOLID *ODBC Driver* 3.5 is Unicode compliant.

Old Client Versions

Old clients can connect to SOLID *Embedded Engine* version 3.5. All Unicode data is converted to ISO Latin 1 whenever possible. Thus, provided only ISO-Latin 1 data is used in the database, old clients can access the database engine.

▶ **Note**

To avoid problems in the future, it is recommended that you upgrade your client applications to use version 3.5 client libraries.

Unicode Variables and Binding

Using string columns containing Unicode data work just like normal character columns. Note that the length of string buffers is given as the number of bytes required to store the value.

String Functions

String functions work as expected, also between ISO Latin 1 and Unicode strings. Conversions are provided implicitly, when necessary. The result is always of Unicode type, if either of the operands is Unicode.

The functions UPPER() and LOWER() work on Unicode strings when the contained characters can be mapped to ISO Latin 1 code page.

Translations

The character translations defined in client side `solid.ini` do not affect the data stored in Unicode columns. Translations remain in effect for character columns.

SOLID *Light Client*

SOLID *Light Client* does not work with Unicode since it does not support any ODBC 3.5 API functionality.

Unicode and SOLID *JDBC Driver*

Unicode is supported in the SOLID *JDBC Driver 2.0*, which is compatible with SOLID *Embedded Engine* 3.0 and 3.5 and SOLID *SynchroNet* 1.1 and 2.0.

As Java uses natively Unicode strings, supporting Unicode means primarily that when accessing Unicode columns in SOLID, no data type conversions are necessary. Additionally, JDBC ResultSet Class methods **getUnicodeStream** and **setUnicodeStream** are supported now for handling large Unicode texts stored in the database engine.

To convert Java applications to support Unicode, the string columns in the database engine need to be redefined with Unicode data types.

5

Using SOLID *Light Client*

This chapter describes how to use SOLID *Light Client*, a very small footprint database client library and a subset of ODBC API, especially designed for implementing embedded solutions with limited memory resources. With SOLID *Light Client*, lightweight client applications can use the full power of SOLID databases.

The topics included in this chapter are:

- What is SOLID *Light Client*?
- Getting started with SOLID *Light Client*
- Running SQL statements on SOLID *Light Client*
- SOLID *Light Client* functions
- Sample code

What is SOLID *Light Client*?

The SOLID *Light Client* library is a 20-function subset of the *ODBC API* (ODBC 1.0 Core), providing full SQL capabilities for application developers accessing SOLID databases. It provides functions for controlling database connections, executing SQL statements, retrieving result sets, committing transactions, and other SOLID functionality. SOLID *Light Client* is suited for target environments with a small amount of memory.

Getting started with SOLID *Light Client*

To get started with SOLID *Light Client*, be sure you have set up the TCP/IP infrastructure as instructed in the installation procedures and your platform specific documentation.

Setting up the Development Environment and Building a Sample Program

Building a program using SOLID *Light Client* library is identical to building any normal C/C++ program:

- Insert the library file to your project.
- Include header file.
- Compile the source code.
- Link the program.

The first two issues are described in more detail in the following sections.

Insert the library file into your project

Check your development environment's documentation on how to link a library to a program. Link the correct *Light Client* library to your program. The libraries are:

Platform	Link the library....
DOS	slcdos35.lib
NT	slcw3235.lib
Solaris	slcssx35.a
VxWorks	slcvxw35.a (ix86) slcvpx35.a (PowerPC)
ChorusOS	slccrx35.z (ix86) slccpx35.a (PowerPC)

Include header files

The following line needs to be included in a *Light Client* program:

```
#include "cli0lcli.h"
```

Insert the directory containing all the other necessary *Light Client* headers into your development environment's include directories setting.

Verifying the Development Environment Setup

The easiest way to verify the development setup is to build a *Light Client* sample program. This enables you to verify your development environment without writing any code. Please note the following that applies to your development environment:

- In the NT environment, the TCP/IP services are provided by standard DLL wsock32.dll. To link these services into your project, add wsock32.lib into linker's lib file list.
- In the NT environment, some development tools link odbc32.lib providing the standard ODBC service as a default library to any project. Because the functions in ODBC have similar names and interfaces as the SOLID *Light Client*, the program may be linked to use ODBC instead of *Light Client*. Remove odbc32.lib from the linker's file list.
- On ChorusOS and VxWorks target machines, you should run a kernal that has a working TCP/IP stack running. Usually you can verify this by checking that the target machine responds to ping requests. For example, if you have configured your target machine to have an IP address 192.168.1.111, you would run "ping 192.168.1.111" from another workstation in your LAN for a response that proves the target is alive:

  ```
  C:\>ping 192.168.1.111
  Pinging 192.168.1.111 with 32 bytes of data:
  Reply from 192.168.1.111: bytes=32 time=260ms TTL=62
  ```

  After verification, your *Light Client* application should work on that target machine.

Connecting to a Database using the Sample Application

Establishing a connection to a database using SOLID *Light Client* library is similar to establishing connections using ODBC. An application needs to obtain an environment handle, allocate space for a connection and establish a connection. Run the sample program to check whether it can obtain a connection to a SOLID database in your environment.

The following code establishes a connection to a SOLID database running in a machine 192.168.1.111 and listening to tcp/ip at port 1313. User account DBA with password DBA has been defined in the database.

```
HENV henv;       /* pointer to environment object                */
```

```
HDBC hdbc;        /* pointer to database connection object  */
RETCODE rc;       /* variable for return code               */

rc = SQLAllocEnv(henv);
if (SQL_SUCCESS != rc)
{
   printf("SQLAllocEnv fails.\n");
   return;
}

rc = SQLAllocConnect(henv,&hdbc);
if (SQL_SUCCESS != rc)
{
   printf("SQLAllocConnect fails.\n");
   return;
}

rc = SQLConnect(hdbc,(UCHAR*)192.168.1.111 1313,SQL_NTS,
(UCHAR*)DBA,SQL_NTS,(UCHAR*)"DBA", SQL_NTS);
if (SQL_SUCCESS != rc)
{
   printf("SQLConnect fails.\n");
   return;
}
```

The connection established above can be cleared using the code below. To make it easier to read no return code checking is included.

```
SQLDisconnect(hdbc);
SQLFreeConnect(hdbc);
SQLFreeEnv(henv);
```

Running SQL Statements on SOLID *Light Client*

This section describes briefly how to do basic database operations with SQL. The following operations are presented here:

- Executing statements through SOLID *Light Client*
- Reading result sets
- Transactions and autocommit mode
- Handling database errors

Executing Statements with SOLID *Light Client*

The code below executes a simple SQL statement `INSERT INTO TESTTABLE (I,C) VALUES (100,'HUNDRED')`. The code expects a valid HENV henv and a valid HDBC hdbc to exist and variable rc of type RETCODE to be defined. The code also expects a table `TESTTABLE` with columns I and C to exist in the database.

```
rc = SQLAllocStmt(hdbc, &hstmt);


if (SQL_SUCCESS != rc)
{
   printf("SQLAllocStmt failed \n");
}
rc = SQLExecDirect(hstmt,(UCHAR*)INSERT INTO TESTTABLE (I,C) VALUES
(100,'HUNDRED'),"SQL_NTS);
if (SQL_SUCCESS != rc)
{
   printf("SQLExecDirect failed \n");
}

rc = SQLTransact(SQL_NULL_HENV,hdbc,SQL_COMMIT);
if ((SQL_SUCCESS != rc))
{
   printf("SQLTransact failed \n");
}

rc = SQLFreeStmt(hstmt,SQL_DROP);
```

```
if ((SQL_SUCCESS != rc))
{
   printf("SQLFreeStmt failed \n");
}
```

Statement with parameters

The code example below prepares a simple statement INSERT INTO TESTTABLE (I,C) VALUES (?,?) to be executed several times with different parameter values. Note, that the *Light Client* does not provide ODBC-like parameter binding. Instead, the values for parameters need to be assigned using the **SQLSetParamValue** function. The following variable definitions are expected:

```
char buf[255];
SDWORD dwPar;
```

As above, the code also expects a valid HENV henv and a valid HDBC hdbc to exist and variable rc of type RETCODE to be defined and a table TESTTABLE with columns I and C to exist in the database.

```
   rc = SQLAllocStmt(hdbc, &hstmt);

   if (SQL_SUCCESS != rc) {
          printf("Alloc statement failed. \n");
   }

   rc = SQLPrepare(hstmt,(UCHAR*)"INSERT INTO TESTTABLE(I,C)
VALUES (?,?)",SQL_NTS);

   if (SQL_SUCCESS != rc) {
         printf("Prepare failed. \n");
   }

   for (i=1;i<100;i++)
   {
         dwPar = i;
         sprintf(buf,"line%i",i);
```

```
    rc = m_lc->LC_SQLSetParamValue(
hstmt,1,SQL_C_LONG,SQL_INTEGER,0,0,&dwPar,NULL );
    if (SQL_SUCCESS != rc) {
            printf("(SetParamValue 1 failed) \n");
            return 0;
    }
    rc = m_lc->LC_SQLSetParamValue(
hstmt,2,SQL_C_CHAR,SQL_CHAR,0,0,buf,NULL );
    if (SQL_SUCCESS != rc) {
            printf("(SetParamValue 1 failed) \n");
            return 0;
    }

    rc = m_lc->LC_SQLExecute(hstmt);

    if (SQL_SUCCESS != rc) {
            printf("SQLExecute failed \n");
    }
    }
    rc = SQLFreeStmt(hstmt,SQL_DROP);
    if ((SQL_SUCCESS != rc)) {
            printf("SQLFreeStmt failed. \n");
    }
```

Reading Result Sets

The following code excerpt prepares the SQL Statement **SELECT I,C FROM TESTTABLE**, executes it and fetches all the rows the database returns. The example code below expects valid definitions for `rc`, `hdbc`, `hstmt`, `henv`.

```
    rc = SQLAllocStmt(hdbc, &hstmt);

    if (SQL_SUCCESS != rc) {
```

```
        printf("SQLAllocStmt failed. \n");
        }

    rc = SQLPrepare(hstmt,(UCHAR*)"SELECT I,C
FROM TESTTABLE",SQL_NTS);

    if (SQL_SUCCESS != rc) {
        printf("SQLPrepare failed. \n");
    }

    rc = SQLExecute(hstmt);

    if (SQL_SUCCESS != rc) {
        printf("SQLExecute failed. \n");
    }

    rc = SQLFetch(hstmt);

if ((SQL_SUCCESS != rc) && (SQL_NO_DATA_FOUND != rc)) {
    printf("SQLFetch returned an unexpected error code . \n");
}

while (SQL_NO_DATA_FOUND != rc)
{
    rc = SQLGetCol(hstmt,1,SQL_C_LONG,&lbuf,sizeof(lbuf),NULL);
    if (SQL_SUCCESS == rc)
    {
          printf("LC_SQLGetCol(1) returns %d \n",lbuf);
          }
          else printf("Error in SQLGetCol(1) \n");
        rc = SQLGetCol(hstmt,2,SQL_C_CHAR,buf,sizeof(buf),NULL);
          if (SQL_SUCCESS == rc)
          {
```

```
                printf("SQLGetCol(2) returns %s \n",buf);
            }
            else printf("Error in SQL_GetCol(2) \n");

            rc = SQLFetch(hstmt);
        }

        rc = m_lc->LC_SQLFreeStmt(hstmt,SQL_DROP);
        if ((SQL_SUCCESS != rc))
        {
            printf("SQLFreeStmt failed. ");
        }
```

Also the following *Light Client* API functions may be useful when processing result sets:

- **SQLDescribeCol**
- **SQLGetCursorName**
- **SQLNumResultCols**
- **SQLSetCursorName**

Transactions and Autocommit Mode

All SOLID *Light Client* connections have the autocommit option set off. There is no method in *Light Client* to set the option on. Every transaction has to be committed explicitly.

To commit the transaction, call the SQLTransact function as follows:

```
rc = SQLTransact(SQL_NULL_HENV,hdbc,SQL_COMMIT);
```

To roll the transaction back, call the SQLTransact as follows.

```
rc = SQLTransact(SQL_NULL_HENV,hdbc,SQL_ROLLBACK);
```

Handling Database Errors

When a *Light Client* API function returns SQL_ERROR or SQL_SUCCESS_WITH_INFO more information about the error or warning can be obtained by calling the **SQLError** function. If the following code is run against a database where no table TESTTABLE is defined, it will produce the appropriate error information.

As usual, the code expects a valid HENV henv and a valid HDBC hdbc to exist and variable `rc` of type `RETCODE` to be defined .

```
    rc = SQLPrepare(hstmt,(UCHAR*)"SELECT I,C FROM 
TESTTABLE",SQL_NTS);

    if (SQL_SUCCESS != rc)
    {
           char buf[255];
           RETCODE rc;

           char szSQLState[255];
           char szErrorMsg[255];
           SDWORD nativeerror = 0;
           SWORD maxerrmsg = 0;

           memset(szSQLState,0,sizeof(szSQLState));
           memset(szErrorMsg,0,sizeof(szErrorMsg));

           rc = SQLError(
           SQL_NULL_HENV,hdbc,hstmt,(UCHAR*)szSQLState,&nativeerror,

(UCHAR*)szErrorMsg,sizeof(szErrorMsg),&maxerrmsg);

           if (SQL_ERROR == rc)
           {
                  printf("SQLError failed \n.");
           }
           else
           {
           printf("Error information dump begins:-------------\n");
                  printf("SQLState '%s' \n",szSQLState);
                  printf("nativeerror %i \n",nativeerror);
                  printf("Errormsg '%s' \n", szErrorMsg);
```

```
            printf("maxerrmsg %i \n",maxerrmsg);
            printf("Error information dump ends:---------------\n");
        }
    }
```

Special Notes about using SOLID *Light Client*

Network Traffic in Fetching Data

SOLID *Light Client* communication does not support SOLID's RowsPerMessage setting. Every *Light Client* call to SQLFetch causes a network message to be sent between client and server. This affects performance when fetching large amounts of data.

Unicode and ODBC Support

SOLID *Light Client* does not work with Unicode and any ODBC 3.5 API functionality. Only ODBC API versions *prior* to 3.5 are supported.

Notes for Programmers Familiar with ODBC

Migrating ODBC Applications to *Light Client API*

If you are using ODBC functions not provided by the *Light Client API*, migrating to SOLID *Light Client* from the standard ODBC database interface requires some programming. Roughly, the migration steps are:

1. Rewiew how your application uses ODBC and estimate whether *Light Client API* functionality is sufficient for you. Some minor changes in your own code are to be expected, basically:
   - Calls to ODBC Extension Level 1 functions should be converted to ODBC Core level functions
   - Rewriting the application without **SQLBindParameter** and **SQLBindCol**
2. Verify your environment using SOLID *Light Client* samples.
3. Modify the ODBC calls in your own code, rebuild and test your program.

SOLID *Light Client* Function Summary

This section lists the functions in SOLID *Light Client API*, which is a subset of the ODBC API. For actual function descriptions, refer to the reference section at the end of this chapter.

Note

SOLID *Light Client* does not provide any ODBC Extension Level functionality for setting parameter values (for example, **SQLBindParameter**) or data binding (for example, SQL-BindCol). Instead SOLID *Light Client* provides SAG CLI compliant functions **SQLSet-ParamValue**, for setting parameter values, and **SQLGetCol**, for reading data from result sets. Read the section, "Non-ODBC SOLID Light Client Functions" for descriptions of these functions.

Summary of Functions

For a complete example program on how to use SOLID *Light Client API*, see "SOLID Light Client Samples" at the end of this section.

Task	Function
Connecting to a data source	*"SQLAllocEnv (ODBC 1.0, Core)"* on page 5-22
	"SQLAllocConnect (ODBC 1.0, Core)" on page 5-21
	"SQLConnect (ODBC 1.0, Core)" on page 5-23
Preparing SQL statements	*"SQLAllocStmt (ODBC 1.0, Core)"* on page 5-22
	"SQLPrepare (ODBC 1.0, Core)" on page 5-35
	"SQLSetParamValue" on page 5-38
	Note this function is unique to SOLID *Client Light*. For details on this function, see the section which follows this table.
	"SQLSetCursorName (ODBC 1.0, Core)" on page 5-37
	"SQLGetCursorName (ODBC 1.0, Core)" on page 5-32
Submitting Requests	*"SQLExecute (ODBC 1.0, Core)"* on page 5-29
	"SQLExecDirect (ODBC 1.0, Core)" on page 5-28

Task	Function
Retrieving Results and Information about Results	*“SQLRowCount (ODBC 1.0, Core)”* on page 5-36 *“SQLNumResultCols (ODBC 1.0, Core)”* on page 5-35 *“SQLDescribeCol (ODBC 1.0, Core)”* on page 5-24 *“SQLGetCol”* on page 5-38 Note that this function is identical to the ODBC compliant function **SQLGetData**. *“SQLFetch (ODBC 1.0, Core)”* on page 5-29 *“SQLGetData (ODBC 1.0, Level 1)”* on page 5-32 Note that this function is identical to its SAG CLI counterpart SQLGetCol. *“SQLError (ODBC 1.0, Core)”* on page 5-27
Terminating a Statement	*“SQLFreeStmt (ODBC 1.0, Core)”* on page 5-31 *“SQLTransact (ODBC 1.0, Core)”* on page 5-37
Terminating a Connection	*“SQLDisconnect (ODBC 1.0, Core)”* on page 5-26 *“SQLFreeConnect (ODBC 1.0, Core)”* on page 5-30 *“SQLFreeEnv (ODBC 1.0, Core)”* on page 5-30

SOLID *Light Client* Samples

Sample 1:

```
#include "sample1.h"

/
**********************************************************************
 *
 * File:           SAMPLE1.C
 *
 * Description:    Sample program for SOLID Light Client API
 *
 * Author:         SOLID
 *
 *
 * SOLID Light Client sample program does the following.
 *
 * 1. Checks that there are enough input parameters to contain
sufficient
 *    connect information
 * 2. Prepares to connect SOLID through Light Client by
 *    allocating memory for HENV and HDBC objects
 * 3. Connects to SOLID using Light Client Library
 * 4. Creates a statement for one query,
 *    'SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM TABLES' for
      reading data from one of SOLID system tables.
 * 5. Executes the query
 * 6. Fetches and outputs all the rows of a result set.
 * 7. Closes the connection gracefully.
 *
 *
**********************************************************************/
void __cdecl main(int argc, char *argv[])
{
```

```
HENV henv;      /* pointer to environment object            */
HDBC hdbc;      /* pointer to database connection object    */
RETCODE rc;     /* variable for return code                 */
HSTMT hstmt;    /* pointer to database statement object     */
char buf[255];  /* buffer for data to be obtained from db   */
char buf2[255]; /* buffer for a printable row to be created */
int iCount = 0; /* counter for rows to be fetched.          */


/* 1. Checks that there are enough input parameters to contain
 /*   sufficient connect information                              */
if (argc != 4)
{
  printf("Proper usage \"connect string\" uid pwd \n");
  printf("argc %i \n",argc);
  return;
}
printf("Will connect SOLID at %s with uid %s and pwd
        %s.\n",argv[1],argv[2],argv[3]);


/* 2. Prepares to connect SOLID through Light Client   /* by
allocating memory for HENV and HDBC objects           */

rc = SQLAllocEnv(&henv);
if (SQL_SUCCESS != rc)
{
  printf("SQLAllocEnv fails.\n");
  return;
}

rc = SQLAllocConnect(henv,&hdbc);
if (SQL_SUCCESS != rc)
{
```

```
    printf("SQLAllocConnect fails.\n");
    return;
  }

  /* 3. Connects to SOLID using Light Client Library */
  rc = SQLConnect(hdbc,(UCHAR*)argv[1],SQL_NTS, (UCHAR*)argv[2],SQL_NTS,
   (UCHAR*)argv[3], SQL_NTS);
  if (SQL_SUCCESS != rc)
  {
    printf("SQLConnect fails.\n");
    return;
  }
  else printf("Connect ok.\n");


  /* 4. Creates a statement for one query,
  /*    data from one of SOLID system tables.                    */

  rc = SQLAllocStmt(hdbc, &hstmt);
  if (SQL_SUCCESS != rc) {
    printf("SQLAllocStmt failed. \n");
   }

  rc = SQLPrepare(hstmt,(UCHAR*)"SELECT
TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM TABLES",SQL_NTS);

  if (SQL_SUCCESS != rc) {
    printf("SQLPrepare failed. \n");
  }
  else printf("SQLPrepare succeeded. \n");


  /* 5. Executes the query */
```

```
rc = SQLExecute(hstmt);
if (SQL_SUCCESS != rc) {
  printf("SQLExecute failed. \n");
}
else printf("SQLExecute succeeded. \n");


/* 6. Fetches and outputs all the rows of a result set. */
rc = SQLFetch(hstmt);
if ((SQL_SUCCESS != rc) && (SQL_NO_DATA_FOUND != rc)) {
  printf("SQLFetch returned an unexpected error code . \n");
}
else printf("Starting to fetch data.\n");

while (SQL_NO_DATA_FOUND != rc)
{
  iCount++;
  sprintf(buf2,"Row %i :",iCount);

  rc = SQLGetCol(hstmt,1,SQL_C_CHAR,buf,sizeof(buf),NULL);
  if (SQL_SUCCESS == rc)
  {
    strcat(buf2,buf);
    strcat(buf2,",");
  }
  else printf("Error in SQL_GetCol(1) \n");

  rc = SQLGetCol(hstmt,2,SQL_C_CHAR,buf,sizeof(buf),NULL);
  if (SQL_SUCCESS == rc)
  {
    strcat(buf2,buf);
    strcat(buf2,",");
  }
```

```
    else printf("Error in SQL_GetCol(2) \n");

    rc = SQLGetCol(hstmt,3,SQL_C_CHAR,buf,sizeof(buf),NULL);
    if (SQL_SUCCESS == rc)
    {
      strcat(buf2,buf);
    }
    else printf("Error in SQL_GetCol(3) \n");

    printf("%s \n",buf2);


    rc = SQLFetch(hstmt);
  }

  rc = SQLFreeStmt(hstmt,SQL_DROP);
  if ((SQL_SUCCESS != rc))
  {
    printf("SQLFreeStmt failed. ");
  }

  /* 7. Closes the connection gracefully.                          */
  SQLDisconnect(hdbc);
  SQLFreeConnect(hdbc);
  SQLFreeEnv(henv);

  printf("Sample program ends successfully.\n");

}
```

Sample 2

```
#ifndef SAMPLE1_H
```

```
#define SAMPLE1_H

/*********************************************************************
 *
 * File:           SAMPLE1.H
 *
 * Description:    Sample program for SOLID Light Client API, header
file
 *
 * Author:         SOLID
 *
 *
 *********************************************************************
/


#include <stdio.h>
#include <string.h>

#include "cli0lcli.h"

#endif
```

Sample 3

```
C:\solid\lcli\samples>sample1 "fb1 1313" DBA DBA
Will connect SOLID at fb1 1313 with uid DBA and pwd DBA.
Connect ok.
SQLPrepare succeeded.
SQLExecute succeeded.
Starting to fetch data.
Row 1 :_SYSTEM,SYS_TABLES,BASE TABLE
Row 2 :_SYSTEM,SYS_COLUMNS,BASE TABLE
Row 3 :_SYSTEM,SYS_USERS,BASE TABLE
```

```
Row 4 :_SYSTEM,SYS_UROLE,BASE TABLE
Row 5 :_SYSTEM,SYS_RELAUTH,BASE TABLE
Row 6 :_SYSTEM,SYS_ATTAUTH,BASE TABLE
Row 7 :_SYSTEM,SYS_VIEWS,BASE TABLE
Row 8 :_SYSTEM,SYS_KEYPARTS,BASE TABLE
Row 9 :_SYSTEM,SYS_KEYS,BASE TABLE
Row 10 :_SYSTEM,SYS_CARDINAL,BASE TABLE
Row 11 :_SYSTEM,SYS_INFO,BASE TABLE
Row 12 :_SYSTEM,SYS_SYNONYM,BASE TABLE
Row 13 :_SYSTEM,TABLES,VIEW
Row 14 :_SYSTEM,COLUMNS,VIEW
Row 15 :_SYSTEM,SQL_LANGUAGES,BASE TABLE
Row 16 :_SYSTEM,SERVER_INFO,VIEW
Row 17 :_SYSTEM,SYS_TYPES,BASE TABLE
Row 18 :_SYSTEM,SYS_FORKEYS,BASE TABLE
Row 19 :_SYSTEM,SYS_FORKEYPARTS,BASE TABLE
Row 20 :_SYSTEM,SYS_PROCEDURES,BASE TABLE
Row 21 :_SYSTEM,SYS_TABLEMODES,BASE TABLE
Row 22 :_SYSTEM,SYS_EVENTS,BASE TABLE
Row 23 :_SYSTEM,SYS_SEQUENCES,BASE TABLE
Row 24 :_SYSTEM,SYS_TMP_HOTSTANDBY,BASE TABLE
Sample program ends successfully.
```

SOLID *Light Client* Function Reference

The following pages describe each ODBC function supported by SOLID *Light Client* in alphabetic order. Each function is defined as a C programming language function.

!

Important

This function reference is specific to ODBC which is a superset of SOLID *Light Client*. Therefore, a function description in this reference may refer to other ODBC functions that do not apply to SOLID *Light Client*. Only the functions listed in the *"SOLID Light Client Function Summary"* on page 5-11 apply to SOLID *Light Client*. In the following descriptions, please disregard any references to non-supported functions.

SQLAllocConnect (ODBC 1.0, Core)

SQLAllocConnect allocates memory for a connection handle within the environment identified by *henv*.

Syntax

RETCODE **SQLAllocConnect**(*henv*, *phdbc*)

The **SQLAllocConnect** function accepts the following arguments.

Type	Argument	Use	Description
HENV	*henv*	Input	Environment handle.
HDBC FAR *	*phdbc*	Output	Pointer to storage for the connection handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

If **SQLAllocConnect** returns SQL_ERROR, it will set the *hdbc* referenced by *phdbc* to SQL_NULL_HDBC. To obtain additional information, the application can call **SQLError** with the specified *henv* and with *hdbc* and *hstmt* set to SQL_NULL_HDBC and SQL_NULL_HSTMT, respectively.

SQLAllocEnv (ODBC 1.0, Core)

SQLAllocEnv allocates memory for an environment handle and initializes the ODBC call level interface for use by an application. An application must call **SQLAllocEnv** prior to calling any other ODBC function.

Syntax

RETCODE **SQLAllocEnv**(*phenv*)

The **SQLAllocEnv** function accepts the following argument.

Type	Argument	Use	Description
HENV FAR *	*phenv*	Output	Pointer to storage for the environment handle.

Returns

SQL_SUCCESS or SQL_ERROR.

If **SQLAllocEnv** returns SQL_ERROR, it will set the *henv* referenced by *phenv* to SQL_NULL_HENV. In this case, the application can assume that the error was a memory allocation error.

SQLAllocStmt (ODBC 1.0, Core)

SQLAllocStmt allocates memory for a statement handle and associates the statement handle with the connection specified by *hdbc*. An application must call **SQLAllocStmt** prior to submitting SQL statements.

Syntax

RETCODE **SQLAllocStmt**(*hdbc*, *phstmt*)

The **SQLAllocStmt** function accepts the following arguments.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.
HSTMT FAR *	*phstmt*	Output	Pointer to storage for the statement handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, or SQL_ERROR.

If **SQLAllocStmt** returns SQL_ERROR, it will set the *hstmt* referenced by *phstmt* to SQL_NULL_HSTMT. The application can then obtain additional information by calling **SQLError** with the *hdbc* and SQL_NULL_HSTMT.

SQLConnect (ODBC 1.0, Core)

SQLConnect loads a driver and establishes a connection to a data source. The connection handle references storage of all information about the connection, including status, transaction state, and error information.

Syntax

RETCODE **SQLConnect**(*hdbc*, *szDSN*, *cbDSN*, *szUID*, *cbUID*, *szAuthStr*, *cbAuthStr*)

The **SQLConnect** function accepts the following arguments.

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
UCHAR FAR *	szDSN	Input	Data source name.
SWORD	cbDSN	Input	Length of *szDSN*.
UCHAR FAR *	szUID	Input	User identifier.
SWORD	cbUID	Input	Length of *szUID*.
UCHAR FAR *	szAuthStr	Input	Authentication string (typically the password).
SWORD	cbAuthStr	Input	Length of *szAuthStr*.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLDescribeCol (ODBC 1.0, Core)

SQLDescribeCol returns the result descriptor — column name, type, precision, scale, and nullability — for one column in the result set; it cannot be used to return information about the bookmark column (column 0).

Syntax

RETCODE **SQLDescribeCol**(*hstmt*, *icol*, *szColName*, *cbColNameMax*, *pcbColName*, *pfSqlType*, *pcbColDef*, *pibScale*, *pfNullable*)

The **SQLDescribeCol** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	hstmt	Input	Statement handle.
UWORD	icol	Input	Column number of result data, ordered sequentially left to right, starting at 1.
UCHAR FAR *	szColName	Output	Pointer to storage for the column name. If the column is unnamed or the column name cannot be determined, the driver returns an empty string.
SWORD	cbColNameMax	Input	Maximum length of the *szColName* buffer.
SWORD FAR *	pcbColName	Output	Total number of bytes (excluding the null termination byte) available to return in *szColName*. If the number of bytes available to return is greater than or equal to *cbColNameMax*, the column name in *szColName* is truncated to *cbColNameMax* – 1 bytes.

SWORD FAR *	pfSqlType	Output	The SQL data type of the column. This must be one of the following values: SQL_BIGINT SQL_BINARY SQL_BIT SQL_CHAR SQL_DATE SQL_DECIMAL SQL_DOUBLE SQL_FLOAT SQL_INTEGER SQL_LONGVARBINARY SQL_LONGVARCHAR SQL_NUMERIC SQL_REAL SQL_SMALLINT SQL_TIME SQL_TIMESTAMP SQL_TINYINT SQL_VARBINARY SQL_VARCHAR or a driver-specific SQL data type. If the data type cannot be determined, the driver returns 0. For more information, see *"SQL Data Types"* on page D-3. For information about driver-specific SQL data types, see the driver's documentation.
UDWORD FAR *	*pcbColDef*	Output	The precision of the column on the data source. If the precision cannot be determined, the driver returns 0.

SWORD FAR *	*pibScale*	Output	The scale of the column on the data source. If the scale cannot be determined or is not applicable, the driver returns 0.
SWORD FAR *	*pfNullable*	Output	Indicates whether the column allows NULL values. One of the following values: SQL_NO_NULLS: The column does not allow NULL values. SQL_NULLABLE: The column allows NULL values. SQL_NULLABLE_UNKNOWN: The driver cannot determine if the column allows NULL values.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLDisconnect (ODBC 1.0, Core)

SQLDisconnect closes the connection associated with a specific connection handle.

Syntax

RETCODE **SQLDisconnect**(*hdbc*)

The **SQLDisconnect** function accepts the following argument.

Type	Argument	Use	Description
HDBC	*hdbc*	Input	Connection handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLError (ODBC 1.0, Core)

SQLError returns error or status information.

Syntax

RETCODE **SQLError**(*henv*, *hdbc*, *hstmt*, *szSqlState*, *pfNativeError*, *szErrorMsg*, *cbErrorMsgMax*, *pcbErrorMsg*)

The **SQLError** function accepts the following arguments.

Type	**Argument**	Use	Description
HENV	*henv*	Input	Environment handle or SQL_NULL_HENV.
HDBC	*hdbc*	Input	Connection handle or SQL_NULL_HDBC.
HSTMT	*hstmt*	Input	Statement handle or SQL_NULL_HSTMT.
UCHAR FAR *	*szSqlState*	Output	SQLSTATE as null-terminated string. For a list of SQLSTATEs, see Appendix A, "ODBC Error Codes."
SDWORD FAR *	*pfNativeError*	Output	Native error code (specific to the data source).
UCHAR FAR *	*szErrorMsg*	Output	Pointer to storage for the error message text.
SWORD	*cbErrorMsgMax*	Input	Maximum length of the *szErrorMsg* buffer. This must be less than or equal to SQL_MAX_MESSAGE_ LENGTH – 1.

SWORD FAR *	*pcbErrorMsg*	Output	Pointer to the total number of bytes (excluding the null termination byte) available to return in *szErrorMsg*. If the number of bytes available to return is greater than or equal to *cbErrorMsgMax*, the error message text in *szErrorMsg* is truncated to *cbErrorMsgMax* – 1 bytes.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLExecDirect (ODBC 1.0, Core)

SQLExecDirect executes a preparable statement, using the current values of the parameter marker variables if any parameters exist in the statement. SQLExecDirect is the fastest way to submit a SQL statement for one-time execution.

Syntax

RETCODE **SQLExecDirect**(*hstmt*, *szSqlStr*, *cbSqlStr*)

The **SQLExecDirect** function uses the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szSqlStr*	Input	SQL statement to be executed.
SDWORD	*cbSqlStr*	Input	Length of *szSqlStr*.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLExecute (ODBC 1.0, Core)

SQLExecute executes a prepared statement, using the current values of the parameter marker variables if any parameter markers exist in the statement.

Syntax

RETCODE **SQLExecute**(*hstmt*)

The **SQLExecute** statement accepts the following argument.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLFetch (ODBC 1.0, Core)

SQLFetch fetches a row of data from a result set. The driver returns data for all columns that were bound to storage locations with **SQLBindCol**.

Syntax

RETCODE **SQLFetch**(*hstmt*)

The **SQLFetch** function accepts the following argument.

Type	Argument	Use	Description
HSTMT	hstmt	Input	Statement handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLFreeConnect (ODBC 1.0, Core)

SQLFreeConnect releases a connection handle and frees all memory associated with the handle.

Syntax

RETCODE **SQLFreeConnect**(*hdbc*)

The **SQLFreeConnect** function accepts the following argument.

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLFreeEnv (ODBC 1.0, Core)

SQLFreeEnv frees the environment handle and releases all memory associated with the environment handle.

Syntax

RETCODE **SQLFreeEnv**(*henv*)

The **SQLFreeEnv** function accepts the following argument.

Type	Argument	Use	Description
HENV	*henv*	Input	Environment handle.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLFreeStmt (ODBC 1.0, Core)

SQLFreeStmt stops processing associated with a specific *hstmt*, closes any open cursors associated with the *hstmt*, discards pending results, and, optionally, frees all resources associated with the statement handle.

Syntax

RETCODE **SQLFreeStmt**(*hstmt*, *fOption*)

The **SQLFreeStmt** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	hstmt	Input	Statement handle
UWORD	fOption	Input	One of the following options: SQL_ CLOSE: Close the cursor associated with *hstmt* (if one was defined) and discard all pending results. The application can reopen this cursor later by executing a **SELECT** statement again with the same or different parameter values. If no cursor is open, this option has no effect for the application. SQL_DROP: Release the *hstmt*, free all resources associated with it, close the cursor (if one is open), and discard all pending rows. This option terminates all access to the *hstmt*. The *hstmt* must be reallocated to be reused. SQL_UNBIND: Release all column buffers bound by **SQLBindCol** for the given *hstmt*. SQL_RESET_PARAMS: Release all parameter buffers set by **SQLBindParameter** for the given *hstmt*.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLGetCursorName (ODBC 1.0, Core)

SQLGetCursorName returns the cursor name associated with a specified *hstmt*.

Syntax

RETCODE **SQLGetCursorName**(*hstmt*, *szCursor*, *cbCursorMax*, *pcbCursor*)

The **SQLGetCursorName** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	hstmt	Input	Statement handle.
UCHAR FAR *	szCursor	Output	Pointer to storage for the cursor name.
SWORD	cbCursorMax	Input	Length of *szCursor.*
SWORD FAR *	pcbCursor	Output	Total number of bytes (excluding the null termination byte) available to return in *szCursor.* If the number of bytes available to return is greater than or equal to *cbCursorMax*, the cursor name in *szCursor* is truncated to *cbCursorMax* – 1 bytes.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLGetData (ODBC 1.0, Level 1)

SQLGetData returns result data for a single unbound column in the current row. The application must call **SQLFetch**, or **SQLExtendedFetch** and (optionally) **SQLSetPos** to position the cursor on a row of data before it calls **SQLGetData**. It is possible to use **SQLBindCol** for some columns and use **SQLGetData** for others within the same row. This function can be used to retrieve character or binary data values in parts from a column with a character, binary, or data source–specific data type (for example, data from SQL_LONGVARBINARY or SQL_LONGVARCHAR columns).

Syntax

RETCODE **SQLGetData**(*hstmt*, *icol*, *fCType*, *rgbValue*, *cbValueMax*, *pcbValue*)

The **SQLGetData** function accepts the following arguments:

Type	Argument	Use	Description
HSTMT	hstmt	Input	Statement handle.
UWORD	icol	Input	Column number of result data, ordered sequentially left to right, starting at 1. A column number of 0 is used to retrieve a bookmark for the row; bookmarks are not supported by ODBC 1.0 drivers or **SQLFetch**.
SWORD	fCType	Input	The C data type of the result data. This must be one of the following values: SQL_C_BINARY SQL_C_BIT SQL_C_BOOKMARK SQL_C_CHAR SQL_C_DATE SQL_C_DEFAULT SQL_C_DOUBLE SQL_C_FLOAT SQL_C_SLONG SQL_C_SSHORT SQL_C_STINYINT SQL_C_TIME SQL_C_TIMESTAMP SQL_C_ULONG SQL_C_USHORT SQL_C_UTINYINT SQL_C_DEFAULT specifies that data be converted to its default C data type. **Note** Drivers must also support the following values of *fCType* from ODBC 1.0. Applications must use these values, rather than the ODBC 2.0 values, when calling an ODBC 1.0 driver: SQL_C_LONG SQL_C_SHORT SQL_C_TINYINT For information about how data is converted, see *"Converting Data from SQL to C Data Types"* on page D-21.
PTR	rgbValue	Output	Pointer to storage for the data.

SDWORD	cbValueMax	Input	Maximum length of the *rgbValue* buffer. For character data, *rgbValue* must also include space for the null-termination byte. For character and binary C data, *cbValueMax* determines the amount of data that can be received in a single call to **SQLGetData**. For all other types of C data, *cbValueMax* is ignored; the driver assumes that the size of *rgbValue* is the size of the C data type specified with *fCType* and returns the entire data value.
SDWORD FAR *	pcbValue	Output	SQL_NULL_DATA, the total number of bytes (excluding the null termination byte for character data) available to return in *rgbValue* prior to the current call to **SQLGetData**, or SQL_NO_TOTAL if the number of available bytes cannot be determined. For character data, if *pcbValue* is SQL_NO_TOTAL or is greater than or equal to *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax* – 1 bytes and is null-terminated by the driver. For binary data, if *pcbValue* is SQL_NO_TOTAL or is greater than *cbValueMax*, the data in *rgbValue* is truncated to *cbValueMax* bytes. For all other data types, the value of *cbValueMax* is ignored and the driver assumes the size of *rgbValue* is the size of the C data type specified with *fCType*.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLNumResultCols (ODBC 1.0, Core)

SQLNumResultCols returns the number of columns in a result set.

Syntax

RETCODE **SQLNumResultCols**(*hstmt*, *pccol*)

The **SQLNumResultCols** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	hstmt	Input	Statement handle.
SWORD FAR *	pccol	Output	Number of columns in the result set.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLPrepare (ODBC 1.0, Core)

SQLPrepare prepares a SQL string for execution.

Syntax

RETCODE **SQLPrepare**(*hstmt*, *szSqlStr*, *cbSqlStr*)

The **SQLPrepare** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	hstmt	Input	Statement handle.
UCHAR FAR *	szSqlStr	Input	SQL text string.
SDWORD	cbSqlStr	Input	Length of *szSqlStr*.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLRowCount (ODBC 1.0, Core)

SQLRowCount returns the number of rows affected by an **UPDATE**, **INSERT**, or **DELETE** statement or by a SQL_UPDATE, SQL_ADD, or SQL_DELETE operation in **SQLSetPos**.

Syntax

RETCODE **SQLRowCount**(*hstmt*, *pcrow*)

The **SQLRowCount** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	hstmt	Input	Statement handle.
SDWORD FAR *	pcrow	Output	For **UPDATE**, **INSERT**, and **DELETE** statements and for the SQL_UPDATE, SQL_ADD, and SQL_DELETE operations in **SQLSetPos**, *pcrow* is the number of rows affected by the request or –1 if the number of affected rows is not available. For other statements and functions, the driver may define the value of *pcrow*. For example, some data sources may be able to return the number of rows returned by a **SELECT** statement or a catalog function before fetching the rows. **Note:** Many data sources cannot return the number of rows in a result set before fetching them; for maximum interoperability, applications should not rely on this behavior.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLSetCursorName (ODBC 1.0, Core)

SQLSetCursorName associates a cursor name with an active *hstmt*. If an application does not call **SQLSetCursorName**, the driver generates cursor names as needed for SQL statement processing.

Syntax

RETCODE **SQLSetCursorName**(*hstmt*, *szCursor*, *cbCursor*)

The **SQLSetCursorName** function accepts the following arguments.

Type	Argument	Use	Description
HSTMT	*hstmt*	Input	Statement handle.
UCHAR FAR *	*szCursor*	Input	Cursor name.
SWORD	*cbCursor*	Input	Length of *szCursor*.

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

SQLTransact (ODBC 1.0, Core)

SQLTransact requests a commit or rollback operation for all active operations on all *hstmts* associated with a connection. **SQLTransact** can also request that a commit or rollback operation be performed for all connections associated with the *henv*.

Syntax

RETCODE **SQLTransact**(*henv*, *hdbc*, *fType*)

The **SQLTransact** function accepts the following arguments.

Type	Argument	Use	Description
HENV	henv	Input	Environment handle.
HDBC	hdbc	Input	Connection handle.

UWORD	fType	Input	One of the following two values: SQL_COMMIT SQL_ROLLBACK

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

Non-ODBC SOLID *Light Client* Functions

This sections describes the two non-ODBC functions supported in SOLID *Light Client*:

- **SQLGetCol**
- **SQLSetParamValue**

SQLGetCol

SQLGetCol gets result data for a single column in the current row. This function allows the application to retrieve the data one column at a time. It may also be used to retrieve large data values in easily manageable blocks.

SQLGetCol functionality is identical to its ODBC API counterpart SQLGetData. For details, read *“SQLGetData (ODBC 1.0, Level 1)”* on page 5-32.

SQLSetParamValue

Sets the value of a parameter marker in the SQL statement specified in **SQLPrepare**. Parameter markers are numbered sequentially from left-to-right, starting with one, and may be set in any order. The value of argument rgbValue will be used for the parameter marker when **SQLExecute** is called.

Syntax

RETCODE **SQLSetParamValue**(*hstmt*, *ipar*, *fCType*, *fSqlType*, *cbColDef*, *ibScale*, *rgb-Value*, *pcbValue*)

The **SQLSetParamValue** function accepts the following arguments:

Type	Argument	Use	Description
HSTMT	hstmt	Input	Statement handle.

UWORD	ipar	Input	Parameter number, ordered squentially left to right, starting at 1.
SWORD	fCType	Input	The C data type of the result data. Check the allowed data type conversions at the end of this chapter. This must be one of the following values: SQL_C_BINARY SQL_C_CHAR SQL_C_DOUBLE SQL_C_FLOAT SQL_C_LONG SQL_C_SHORT
SDWORD	fSqlType	Input	The SQL data type of the parameter. Check the allowed data type conversions following this table. This must be one of the following values: SQL_C_BINARY SQL_C_CHAR SQL_DATE SQL_DECIMAL SQL_C_DOUBLE SQL_C_FLOAT SQL_INTEGER SQL_LONGVARBINARY SQL_LONGVARCHAR SQL_NUMERIC SQL_REAL SQL_SMALLINT SQL_TIME SQL_TIMESTAMP SQL_TINYINT SQL_VARBINARY SQL_VARCHAR
UDWORD	cbColDef	Input	The precision of the column or expression of the corresponding parameter marker.
SWORD	ibScale	Input	The scale of the column or expression of the corresponding parameter marker.
PTR	rgbValue	Input	Output data.
SDWORD *	pcbValue	Input	Length of data in rgbValue

fCType describes the contents of rgbValue. fCType must either be SQL_C_CHAR ot the C equivalent of argument fSqlType. If fCType is SQL_C_CHAR and fSqlType is a numeric type, rgbValue will be converted from a character string to the type specified by fSqlType.

fSqlType is the data type of the column or expression referenced by the parameter marker. At execute time, the value in rgbValue will be read and converted from fCType to fSqlType, and then sent to the SOLID database. Note that the value of rgbValue remains unchanged.

cbColDef is the length or precision of the column definition for the column or expression referenced. cbColDef differs depending on the class of data as follows:

Type	Description
SQL_CHAR SQL_VARCHAR	maximum length of the column
SQL_DECIMAL SQL_NUMERIC	maximum decimal precision (that is, total number of digits possible)

ibScale is the total number of digits to the right of the decimal point for the column referenced. ibScale is defined only for the SQL_DECIMAL and SQL_NUMERIC data types. rgbValue is a character string that must contain the actual data for the parameter marker. The data must be of the form specified by the fCType argument.

pcbValue is an integer that is the length of the parameter marker value in rgbValue. It is only used when fCType is SQL_C_CHAR or when specifying a null database value. The variable must be set to SQL_NULL_DATA if a null value is to be specified for the parameter marker. If the variable is set to SQL_NTS then rgbValue will be treated as a null terminated string.

Returns

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE.

Diagnostics

- If the data identified by the fcType argument cannot be converted to the data value identified by the fSqlType argument, SQL_ERROR is returned ('07006' -- Restricted data type attribute violation)
- If the fcType argument is not valid, SQL_ERROR is returned ('S1003' -- Program type out of range).
- If the fSqlType argument is not valid, SQL_ERROR is returned ('S1004' -- SQL data type out of range).

- If the ipar argument is less than 1, SQL_ERROR is returned ('S1009' -- Invalid argument value).

Comments

All parameters set by this function remain in effect until either **SQLFreeStmt** is called with the SQL_UNBIND_PARAMS or SQL_DROP option or **SQLSetParamValue** is called again for the same parameter number. When a SQL statement containing parameters is executed, the set values of the parameters are sent to to the SOLID database.

Note that the number of parameters must match exactly the number of parameter markers present in the statement that was prepared. If less parameter values are set than there were parameter markers in the SQL statement, NULL values will be used instead.

Code Example

The code example below prepares a simple statement INSERT INTO TESTTABLE (I,C) VALUES (?,?) to be executed several times with different parameter values.

```
...
    char buf[255];
    SDWORD dwPar;
...
    rc = SQLPrepare(hstmt,(UCHAR*)"INSERT INTO TESTTABLE(I,C)
    VALUES (?,?)",SQL_NTS);
    if (SQL_SUCCESS != rc) {
          printf("Prepare failed. \n");
    }
    for (i=1;i<100;i++)
    {
          dwPar = i;
          sprintf(buf,"line%i",i);

            rc = m_lc->LC_SQLSetParamValue(
hstmt,1,SQL_C_LONG,SQL_INTEGER,0,0,&dwPar,NULL );
            if (SQL_SUCCESS != rc) {
                  printf("(SetParamValue 1 failed) \n");
                  return 0;
```

```
          }

          rc =
        m_lc->LC_SQLSetParamValue(
hstmt,2,SQL_C_CHAR,SQL_CHAR,0,0,buf,NULL );
          if (SQL_SUCCESS != rc) {
                printf("(SetParamValue 1 failed) \n");
                return 0;> >
         }
```

Related Functions

For information about	See
Preparing a statement for execution	SQLPrepare
Executing a prepared SQL statement	SQLExecute
Executing a SQL statement	SQLExecDirect

SOLID *Light Client* Type Conversion Matrix

The table below describes the type conversions provided by the SOLID *Light Client* functions **SQLGetCol** and **SQLSetParamValue**.

Abbreviations used in the tables for the C variable data types are as follows:

Abbreviation	API parameter definition	C variable data types
Bin	SQL_C_BINARY	voidd*
Char	SQL_C_CHAR	char[], char*
Long	SQL_C_LONG	long int (*), 32 bits
Short	SQL_C_SHORT	short int (*), 16 bits
Float	SQL_C_FLOAT	float (*)
Double	SQL_C_DOUBLE	double (*)

(*) Note that when variables of these data types are used as parameters in *Light Client* functions calls, actually the pointer to the variable must be passed instead.

Refer to *Appendix D, "Data Types"* for a description of SQL data types.

Functions **SQLGetCol** and **SQLGetData** perform the following data type conversions between database column types and C variable data types:

SQL data type \ C variable data type	Bin	Char	Long	Short	Float	Double
TINYINT	*	*	*	*	*	*
LONG VARBINARY	*	*				
VARBINARY	*	*				
BINARY	*	*				
LONG VARCHAR	*	*				
CHAR	*	*				
NUMERIC		*	*	*	*	*
DECIMAL		*	*	*	*	*
INTEGER	*	*	*	*	*	*

SQL data type \ C variable data type	Bin	Char	Long	Short	Float	Double
SMALLINT	*	*	*	*	*	*
FLOAT	*	*	*	*	*	*
REAL	*	*	*	*	*	*
DOUBLE	*	*	*	*	*	*
DATE		*				
TIME		*				
TIMESTAMP		*				
VARCHAR	*	*				

Function **SQLSetParamValue** provides the following type conversions between C data types and the database column types.

SQL data type \ C variable data type	Bin	Char	Long	Short	Float	Double
TINYINT		*	*	*		
LONG VARBINARY	*					
VARBINARY	*					
BINARY	*					
LONG VARCHAR		*				
CHAR		*				
NUMERIC		*	*	*	*	*
DECIMAL		*	*	*	*	*
INTEGER		*	*	*		
SMALLINT		*	*	*		
FLOAT		*	*	*	*	*
REAL		*	*	*	*	*
DOUBLE		*	*	*	*	*
DATE		*				

SQL data type \ C variable data type	Bin	Char	Long	Short	Float	Double
TIME		*				
TIMESTAMP		*				
VARCHAR		*				

6

Using the SOLID *JDBC Driver*

This chapter describes how to use the SOLID *JDBC Driver,* a 100% Pure Java^{TM} implementation of the Java Database Connectivity (JDBC^{TM}) standard. The chapter covers the following information:

- What is SOLID *JDBC Driver*?
- Getting started with SOLID *JDBC Driver*
- Running SQL statement with SOLID *JDBC Driver*
- Connecting a Solid server through JDBC
- SOLID *JDBC Driver* interfaces and methods
- Sample code

What is SOLID *JDBC Driver*?

The JDBC API, Java API's core API for JDK 1.2, defines Java classes to represent database connections, SQL statements, result sets, database metadata, etc. It allows a Java programmer to issue SQL statements and process the results. JDBC is the primary API for database access in Java.

JDBC drivers can either be entirely written in Java so that they can be downloaded as part of an applet, or they can be implemented using native methods to bridge to existing database access libraries. SOLID *JDBC Driver* provides Java developers with native database access to Solid servers. SOLID *JDBC Driver* is written entirely in Java and communicates to a SOLID database server through SOLID's native network protocol.

SOLID *JDBC Driver 2.0* can be downloaded quickly (with a compact bytecode of 49 KB), enabling efficient SOLID database use in thin-client Java applications. It offers JDBC standard compliance and is 100% pure Java certified. It is usable in all Java environments supporting JDK 1.2. The SOLID *JDBC Driver 2.0* is compatible with SOLID *Embedded Engine* 3.0 and 3.5 and SOLID *SynchroNet* 1.1 and 2.0.

Getting started with SOLID *JDBC Driver*

To get started with SOLID *JDBC Driver*, be sure you have:

1. Installed the *JDBC Driver* and verified the installation. For details, follow the instructions on the SOLID *JDBC Driver* Web site.
2. Set up the development environment so that it support JDBC properly. SOLID *JDBC Driver* expects support for JDBC version 2.0x. The JDBC interface is included in the `java.sql` package. To import this package, be sure to include the following line in the application program:

```
import java.sql.*;
```

Registering SOLID *JDBC Driver*

The JDBC driver manager, which is written entirely in Java, handles loading and unloading drivers and interfacing connection requests with the appropriate driver. It was Java API's intention to make the use of a specific JDBC driver as transparent as possible to the programmer and user. The driver can be registered with the three alternative ways, which are shown below. The parameter required **byClass.forName** and **Properties.put** functions is the name of the driver, which is **solid.jdbc.SolidDriver**.

```
// registration using Class.forName service
Driver)Class.forName("solid.jdbc.SolidDriver")
// a workaround to a bug in some JDK1.1
implementations
Driver d =
(Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();

// Registration using system properties
variable also
Properties p = System.getProperties();
p.put("jdbc.drivers","solid.jdbc.SolidDriver");
System.setProperties(p);
```

See the source code for the Sample 1 application in *"Code Examples"* on page 6-27.

Connecting to the Database

Once the driver is successfully registered with the driver manager a connection is established by creating a Java Connection object with the following code. The parameter required by the **DriverManager.getConnection** function is the JDBC connection string.

```
Connection conn = null;
try {
    conn =  DriverManager.getConnection(sCon);
}
catch (Exception e) {
    System.out.println("Connect failed : " +
e.getMessage());
    throw new Exception("Halted.");
}
```

The connect string structure is jdbc:solid://*machine name*:*port*/*user name*/*password*. The string "jdbc:solid://fb9:1314/dba/dba" attempts to connect a Solid server in machine fb9 listening tcp/ip protocol at port 1314.

The application can establish several Connection objects to database. Connections can be closed be the following code.

```
conn.close();
```

See the source code for the Sample 1 application in *"Code Examples"* on page 6-27.

Running SQL Statements with JDBC

This section describes briefly how to do basic database operations with SQL. The following operations are presented here:

- Executing statements through JDBC
- Reading result sets
- Transactions and autocommit mode
- Handling database errors
- Using **DatabaseMetadata**

For more detailed description on these subjects, refer also to JDBC documentation.

Executing a Simple Statement

The following code expects that a *Connection* object *conn* is established before calling the code.

```
stmt= conn.createStatement();
stmt.execute("INSERT INTO JDB_TEST (I1,I2)
VALUES (2,3)");
```

Note

The insert is not committed by the code unless the database is in autocommit mode.

See the source code for the Sample 1 application in *"Code Examples"* on page 6-27.

Statement with Parameters

The code below creates a `PreparedStatement` object for a query, assigns values for its parameters and executes the query. Check the available methods for setting values to different column types from the *"SOLID JDBC Driver Type Conversion Matrix"* on page 6-50. The code expects a `Connection` object `conn` to be established.

```
PreparedStatement pstmt;
int count, cnt;
int i;

sQuery = "INSERT INTO ALLTYPES
(TI,SI,II,RR,FF,DP,DE,NU,CH,VC,DT,TM,TS) VALUES";
sQuery = sQuery + "(?,?,?,?,?,?,?,?,?,?,?,?,?)";

pstmt= conn.prepareStatement(sQuery);
pstmt.setInt(1,101);
pstmt.setInt(2,102);
pstmt.setInt(3,103);
pstmt.setDouble(4,2104.56);
pstmt.setDouble(5,104.56);
pstmt.setDouble(6,3104.56);
```

```
pstmt.setDouble(7,204.56);
pstmt.setDouble(8,304.56);
pstmt.setString(9,"cccc");
pstmt.setString(10,"longer string");

java.sql.Time pTime = new
java.sql.Time(11,11,11);
java.sql.Date pDate = new java.sql.Date(96,1,2);

java.sql.Timestamp pTimestamp = new
java.sql.Timestamp(96,1,2,11,11,11,0);
pstmt.setDate(11,pDate);
pstmt.setTime(12,pTime);
pstmt.setTimestamp(13,pTimestamp);

pstmt.executeUpdate();
```

See the source code for the Sample 3 application in *"Code Examples"* on page 6-27.

Note

The insert is not committed by the code unless the database is in autocommit mode.

Reading result sets

The code below obtains a result set for the SQL and prints out column name and type information for each column in the result set using the `ResultSetMetaData` object.

```
SELECT TABLE_CATALOG,TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM
SYS_TABLES WHERE ID < 10000
```

The code then loops through the result set and prints the data in each column in each row by using `getString` method. Check the available methods for accessing data of different column types from the *"SOLID JDBC Driver Type Conversion Matrix"* on page 6-50. The code expects a `Connection` object `conn` to be established.

```
String sQuery;
```

```
ResultSetMetaData meta;
Statement stmt;
ResultSet result;
int count, cnt;
int i;

// the query to be executed
sQuery = "SELECT
TABLE_CATALOG,TABLE_SCHEMA,TABLE_NAME," ;
sQuery = sQuery + "TABLE_TYPE FROM
SYS_TABLES WHERE ID < 10000";

// we create statement for the query
stmt= conn.createStatement();
// execute it and obtain a result set
result = stmt.executeQuery(sQuery);

// to see what we got we obtain a
ResultSetMetaData object meta = result.getMetaData();
// check the number of columns
count = meta.getColumnCount();

// print some information about the columns
for (i=1; i &lt;= count; i++)
{
   String sName = meta.getColumnName(i);
   int iType = meta.getColumnType(i);
   String sTypeName = meta.getColumnTypeName(i);

   System.out.println("Col:"+i+" "+sName+ "," + iType + "," +
sTypeName);
}
```

```
// and finally, loop through the ResultSet and
print the data out
int cnt = 1;
while(result.next())
{
   for (i=1; i &lt;= count; i++)
   {
     System.out.println("Row:"+cnt+ " column:" +i+"
: "+result.getString(i));
   }
   cnt++;
}
```

Note

It is possible to improve the performance of reading large result sets by instructing a Solid server to return several rows of the result set in one network message. This functionality is activated by editing configuration `RowsPerMessage` in section [Srv] in a Solid server configuration file *solid.ini*. The default value is 10. This is new functionality in *JDBC Drive*r 2.3. In prior versions, the rows of the result set were always returned one by one.

See the source code for the Sample 1 application in *"Code Examples"* on page 6-27.

Transactions and Autocommit Mode

A SOLID database can be in either autocommit or non-autocommit mode. When not in autocommit mode each transaction needs to be explicitly committed before the modifications it made can be seen to other database connections. The autocommit state can be monitored by `Connection.getAutoCommit()` function. The state can be set by `Connection.setAutoCommit()`. A Solid server's default setting for autocommit state is `true`. If autocommit mode is off the transactions can be committed in two ways.

- using Connection.commit() function or
- executing a statement for SQL 'COMMIT WORK'

Handling Database Errors

In some cases it is necessary for the application to recover from a database error. For example, a unique key constraint violation can be recovered by assigning the row a different key. The code below expects a `Statement` object stmt to exist and `String sQuery` to contain SQL that may cause an error. A database native error code will be assigned to variable `ec`. For native error codes, see the appendix, "Error Codes, in the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide**.

```
try {
    result = stmt.executeQuery(sQuery);


}
catch (SQLException e) {
    int ec = e.getErrorCode();
    String ss = e.getSQLState();
    String s2 = e.toString();
    System.out.println("Native error code:" + ec);
}
```

Using DatabaseMetadata

Interface `DatabaseMetaData` contains information about the database behind the connection. Usually this information is necessary for application development tools not actual applications. If you are developing an application on JDBC interface for one kind of database engine this is seldom if ever necessary. If you are developing an application to run on several database engines the application can obtain necessary information about the database through `DatabaseMetaData`.

A `DatabaseMetaData` object can be obtained from the `Connection` object by the code below. The code also extracts database product name to string sName and all the views in the database to `ResultSet rTables`. As usual, the code expects that a `Connection` object `conn` is established before calling it.

```
DatabaseMetaData meta;


String sName;
ResultSet rTables;


String types[] = new String[1];
```

```
types[0] = "VIEW";

meta = conn.getMetaData();
sName = meta.getDatabaseProductName();
rTables =
meta.getTables(null,"","",types);
```

Special Notes About SOLID and JDBC

JDBC does not really specify what SQL you can use; it simply passes the SQL on to the driver and lets the driver either pass it on directly to the database, or parse the SQL itself. Because of this, the SOLID *JDBC Driver* behavior is particular to the SOLID database. In some functions the JDBC specification leaves some details open. Check *"JDBC Driver Interfaces and Methods"* on page 6-10 for the details particular to SOLID's implementation of the methods.

Executing stored procedures

In a SOLID database, stored procedures can be called by executing statements 'CALL proc_name [parameter ...] ' as in any other SQL. Procedures are thus used in JDBC in the same way as any statement.

Note

SOLID stored procedures can return result sets. Calling procedures through JDBC CallableStatement interface is not necessary. For an example of calling SOLID procedures using JDBC, see the source code for the Sample 3 application in *"Code Examples"* on page 6-27.

Interface CallableStatement

A JDBC `CallableStatement` interface is intended to support calling database stored procedures. The interface is not necessary when writing applications on a Solid server. Portability reasons, for instance, can make using `CallableStatement` a good decision. The example below illustrates running simple SQL statements using this interface.

```
CallableStatement csta;
int i1,i2;
String s1;
```

```
ResultSet res;

// creating a CallableStatement object
csta = conn.prepareCall("select * from
keytest where i1 = ?");

// assigning a value for parameter
csta.setInt(1,1);

// obtaining a result set
res = csta.executeQuery();

while (res.next())
{
   i1 = csta.getInt(1);
   i2 = csta.getInt(2);
   s1 = csta.getString(3);
   System.out.println("Row contains " + i1 + "," + i2 +
"," + s1);
}
```

JDBC Driver Interfaces and Methods

This section lists the Java interfaces contained by the SOLID *JDBC Driver* and their methods. JDBC is a standard application interface for databases. Sun provides the official documentation of JDBC interface classes and methods at the following Web site:

`http://java.sun.com/products/jdk/1.2/docs/index.html`

SOLID *JDBC Driver* conforms to the JDBC standard and thus SOLID will neither repeat nor maintain the JDBC interface documentation. Instead, this section lists all behavior specific to SOLID *JDBC Driver* and a Solid server.

For a description of how different data types are supported by SOLID *JDBC Driver*, see the JDBC Driver Type Conversion Matrix at the end of this chapter.

Array

The java.sql.Array interface is not supported. This interface is used to map SQL type Array in the Java programming language. It reflects a SQL3 standard that is currently unavailable in the SOLID database.

Blob

The java.sql.Blob interface is not supported. This interface is used to map SQL type Blob in the Java programming language. It reflects a SQL3 standard that is currently unavailable in the SOLID database.

CallableStatement

A java.sql.`CallableStatement` interface is intended to support calling database stored procedures. Thus, SOLID stored procedures are used in JDBC in the same way as any statement; the use of class `CallableStatement` is not necessary when you are writing applications on a Solid server only. However, for portability reasons, using `CallableStatement` is a wise choice.

Methods

Method name	Notes
getArray(int i)	Supports a SQL3 standard that is currently unavailable in the Solid database.
getBigDecimal(int parameterIndex)	Works as specified in Java API.
getBigDecimal (int parameterIndex, int scale)	Deprecated.
getBlob(int i)	Works as specified in Java API.
getBoolean(int parameterIndex)	Works as specified in Java API.
getByte(int parameterIndex)	Works as specified in Java API.
getBytes(int parameterIndex)	Works as specified in Java API.
getClob(int i)	Works as specified in Java API.
getDate(int parameterIndex)	Works as specified in Java API.
getDate(int parameterIndex, calendar cal)	Works as specified in Java API.
getDouble(int parameterIndex)	Works as specified in Java API.
getFloat(int parameterIndex)	Works as specified in Java API.

getInt(int parameterIndex)	Works as specified in Java API.
getLong(int parameterIndex)	Works as specified in Java API.
getObject(int parameterIndex)	Works as specified in Java API.
getObject (int i, Map map)	Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"
getRef(int i)	Supports a SQL3 standard that is currently unsupported in the SOLID database.
getShort(int parameterIndex)	Works as specified in Java API.
getString(int parameterIndex)	Works as specified in Java API.
getTime(int parameterIndex)	Works as specified in Java API.
getTimestamp(int parameterIndex, Calendar cal)	Works as specified in Java API.
registerOutParameter(int, parameterIndex, int sqlType)	Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"
registerOutParameter(int parameterIndex, int sqlType, int scale)	Not supported by SOLID. Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"
registerOutParameter(int parameterIndex, int sqlType, String typeName)	Not supported by SOLID. Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"
wasNull()	Not supported by SOLID. Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"

Clob

The java.sql.Clob interface is not supported. This interface is used to map SQL type Clob in the Java programming language. It reflects a SQL3 standard that is currently unavailable in the SOLID database.

Connection

The java.sql.Connection interface is a public interface. It is used to establish a connection (session) with a specified database. SQL statements are executed and results are returned within the context of a connection.

Method name	Notes
clearWarnings()	Works as specified in Java API.
close()	Works as specified in Java API. Note that connections should be explicitly closed when not used anymore.
commit()	Works as specified in Java API.
CreateStatement()	Works as specified in Java API.
CreateStatement(int resultSetType, int resultSet-Concurency)	The argument resultsetConcurrency is ignored; this is not supported by the SOLID database.
getAutoCommit()	Works as specified in Java API.
getCatalog()	Not supported by SOLID.
getMetaData()	Works as specified in Java API.
getTransactionIsolation()	Works as specified in Java API.
getTypeMap()	Supports a SQL3 standard that is currently unavailable in the SOLID database.
getWarnings()	Works as specified in Java API.
isClosed()	Works as specified in Java API.
isReadOnly()	SOLID only supports read-only database and read-only transactions if the database is declared as read-only. This method always returns false.
nativeSQL(String sql)	Works as specified in Java API. SOLID *JDBC Driver* does not change the SQL passed to the Solid server. The SQL query the user passes is returned.
prepareCall(String sql)	Works as specified in Java API. Note that the escape call syntax is not supported.

prepareCall(String sql, int resultSetType, int resultSetConcurrency)	The argument resultsetConcurrency is ignored;this is not supported by the SOLID database.
prepareStatement(String sql)	Works as specified in Java API.
prepareStatement(String sql, int resultSetype, int resultSetConcurrency)	The argument resultsetConcurrency is ignored;this is not supported by the SOLID database.
rollback()	Works as specified in Java API.
setAutoCommit(boolean autoCommit)	Works as specified in Java API.
setCatalog(String catalog)	Works as specifed by Java API.
setReadOnly(boolean readOnly)	Solid only supports read-only database and read-only transactions if the database is declared as read-only.This method exists but does not affect the connection behavior.
setTransactionIsolation(int level)	Works as specified in Java API.
setTypeMap(Map map)	Supports a SQL3 standard that is currently unavailable in the SOLID database.

DatabaseMetaData

The java.sql.DatabaseMetaData interface is a public abstract database. It provides general, comprehensive information about the database.

All method for this interface are supported by SOLID, *except*:

- getColumnPrivileges(String catalog, String schema, String table, string columnNamePattern)
- getUDTs(String catalog, String schemaPattern, String typeNamePattern, int [] types)

Note that the following SQL datatypes are not supported: ARRAY, BLOB, CLOB, DISTINCT, JAVA_OBJECT, OTHER, REF, and STRUCT.

Driver

The java.sql.Driver interface is a public abstract interface. Every driver class implements this interface.

Method name	Notes
acceptsURL(String url)	Works as specified in Java API.
connect(String url, Properties info)	Works as specified in Java API.
getMajorVersion()	Works as specified in Java API.
getMinorVersion()	Works as specified in Java API.
getPropertyInfo(String url, Properties info)	Works as specified in Java API.
jdbcCompliant()	Works as specified in Java API. Returns ’Yes’ as boolean.

PreparedStatement

The java.sql.PreparedStatement interface is a public abstract interface. It extends the Statement interface. It provides an object that represents a precompiled SQL statement.

Subinterfaces:

CallableStatement

Methods

Method name	Notes
addBatch	Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"
clearParameters()	Works as specified in Java API.
execute()	Works as specified in Java API.
executeQuery()	Works as specified in Java API.
executeUpdate()	Works as specified in Java API.
getMetaData()	Works as specified in Java API.
setArray(int i, Array x)	Not supported by SOLID. Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"

setAsciiStream(int parameterIndex, InputStream s, int length)	Works as specified in Java API.
setBigDecimal(int parameterIndex, BigDecimal x)	Works as specified in Java API.
setBinaryStream(int parameterIndex, InputStream x, int length)	Works as specified in Java API.
setBlob(int I, Blob x)	Works as specified in Java API.
setBoolean(int parameterIndex, boolean x)	Works as specified in Java API.
setByte(int parameterIndex, byte x)	Works as specified in Java API.
setBytes(int parameterIndex, byte[] x)	Works as specified in Java API.
setCharacterStream(int parameterIndex, Reader reader, int length	Works as specified in Java API.
setClob(int I, Clob x)	Works as specified in Java API.
setDate(int parameterIndex, Date x)	Works as specified in Java API.
setDate(int parameterIndex, Date x, Calendar cal)	Works as specified in Java API.
setDouble(int parameterIndex, double x)	Works as specified in Java API.
setFloat(int parameterIndex, float x)	Works as specified in Java API.
setInt(int parameterIndex, int x)	Works as specified in Java API.
setLong(int parameterIndex, long x)	Works as specified in Java API.
setNull(int parameterIndex, int sqlType)	Works as specified in Java API.
setNull(int paramIndex, int sqlType, String typeName)	Supports a SQL3 standard that is currently unavailable in the Solid database.
setObject(int parameterIndex, Object x)	Works as specified in Java API.
setObject(int parameterIndex, Object x, int targetSqlType))	Works as specified in Java API. Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"
setObject(int parameterIndex, Object x, int targetSQLType, int scale)	Works as specified in Java API. Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"

setRef(int I, Ref x)	Supports a SQL3 standard that is currently unavailable in the Solid database.
setShort(int parameterIndex, short x, short)	Works as specified in Java API.
setString(int parameterIndex, String x)	Works as specified in Java API.
setTime(int parameterIndex, Time x)	Works as specified in Java API.
setTime(int parameterIndex, Time x Calendar cal)	Works as specified in Java API.
setTimestamp(int parameterIndex, Timestamp x)	Works as specified in Java API.
setTimestamp(int parameterIndex, Time x, Calendar cal)	Works as specified in Java API.
setUnicodeStream(int parameterIndex, InputStream x, int) length	Deprecated.

Ref

The java.sql.Ref interface is a public abstract interface.

This interface is a reference to a SQL structured type value in the database. A Ref can be saved to persistent storage. A Ref is de-referenced by passing it as a parameter to a SQL statement and executing the statement.

Note

This interface supports SQL3. SQL3 data types such as binary large objects, and structured types, are part of JDBC 2.0 API. This API incorporates a model of the new SQL3 types that includes only those properties that are essential to exchanging data between Java applications and databases. The new SQL3 types are not supported by SOLID.

ResultSet

The java.sql.ResultSet interface is a public abstract interface. It is a table of data that represents a database result set from a query statement. This object includes a cusor that points to its current row of data. The cursor's initial position is before the first row. It is moved to the next row by the **next** method. When there are no more rows left in the result set, the object returns false; this allows the use of a WHILE loop to iterate through the result set.

A default resultset object is not updatable and its cursor moves forward only. In JDBC 2.0 API, you can produce result sets that are updatable. For methods, see *"ResultSet (updatable)"* on page 6-25.

Methods

Method name	Notes
absolute(int row)	Works as specified in Java API.
afterLast()	Works as specified in Java API.
beforeFirst	Works as specified in Java API.
CancelRowUpdates()	Not supported by SOLID.
clearWarnings()	Works as specified in Java API.
close()	Works as specified in Java API.
deleteRow()	Works as specified in Java API.
findColumn(String columnName)	Works as specified in Java API.
first()	Works as specified in Java API.
getArray(int i)	Supports a SQL3 standard that is currently unavailable in the SOLID database.
getArray(String ColName)	Supports a SQL3 standard that is currently unavailable in the SOLID database.
getAsciiStream(int columnIndex)	Works as specified in Java API.
setAsciiStream(String columnName)	Works as specified in Java API.
getBigDecimal(int columnIndex)	Works as specified in Java API.
getBigDecimal(int columnIndex, int scale)	Deprecated.
getBigDecimal(String columnName)	Works as specified in Java API.
getBigDecimal(String columnName, int scale)	Deprecated.
getBinaryStream(int columnIndex)	Works as specified in Java API.
getBinaryStream(String columnName)	Works as specified in Java API.
getBlob(int I)	Works as specified in Java API.
getBlob(String colName)	Works as specified in Java API.
getBoolean(string columnName)	Works as specified in Java API.

getByte(int columnIndex)	Works as specified in Java API.
getByte(String columnName)	Works as specified in Java API.
getByte(int columnIndex))	Works as specified in Java API.
getBytes(String columnName)	Works as specified in Java API.
getCharacterStream(int columnIndex)	Works as specified in Java API.
getCharacterStream(String columnName)	Works as specified in Java API.
getClob(int I)	Supports a SQL3 standard that is currently unavailable in the SOLID database.
getClob(String colName)	Supports a SQL3 standard that is currently unavailable in the SOLID database.
getConcurrency ()	Not supported by SOLID.
getCursorName()	Works as specified in Java API.
getDate(int columnIndex)	Works as specified in Java API.
getDate(int columnIndex, Calendar cal)	Works as specified in Java API.
getDate(String columnName)	Works as specified in Java API.
getDate(String columnName, Calendar cal)	Works as specified in Java API.
getDouble(int columnIndex)	Works as specified in Java API.
getDouble(String columnName)	Works as specified in Java API.
getFetchDirection()	Works as specified in Java API.
getFetchSize()	No operation in SOLID. The set value a user sets with this method (which is ignored) is returned.
getFloat(int columnIndex)	Works as specified in Java API.
getFloat(String columnName)	Works as specified in Java API.
getInt(int columnIndex)	Works as specified in Java API.
getInt(String columnName)	Works as specified in Java API.
getLong(String columnName)	Works as specified in Java API.
getMetaData()	Works as specified in Java API.
getObject(int columnIndex)	Works as specified in Java API.

getObject(int i, Map map)	Not supported by SOLID. This method throws an exception with the following message: "This method is not supported"
getObject(String columnName)	Works as specified in Java API.
getObject(String colName, Map map)	Not supported by SOLID.This method throws an exception with the following message: "This method is not supported"
getRef(int i)	Supports a SQL3 standard that is currently unavailable in the SOLID database.
getRef(String colName)	Supports a SQL3 standard that is currently unavailable in the SOLID database.
getRow()	Works as specified in Java API.
getShort(int columnIndex)	Works as specified in Java API.
getShort(String columnName)	Works as specified in Java API.
getStatement()	Works as specified in Java API.
getString(int columnIndex)	Works as specified in Java API.
getString(String columnName)	Works as specified in Java API.
getTime(int columnIndex)	Works as specified in Java API.
getTime(int columnIndex, Calendar cal)	Works as specified in Java API.
getTimestamp(String columnName)	Works as specified in Java API.
getTimestamp(String columnName, Calendar cal)	Works as specified in Java API.
getType()	Works as specified in Java API.
getUnicodeStream(int columnIndex)	Deprecated.
getUnicodeStream(String columnName)	Deprecated
getWarnings()	Works as specified in Java API.
insertRow()	Works as specified in Java API.
isAfterLast()	Works as specified in Java API.
isBeforeFirst()	Works as specified in Java API.
isFirst()	Works as specified in Java API.
isLast()	Works as specified in Java API.

last()	Works as specified in Java API.
moveToCurrentRow()	Works as specified in Java API.
moveToInsertRow()	Works as specified in Java API.
next()	Works as specified in Java API.
previous()	Works as specified in Java API.
refreshRow()	Not supported by SOLID.
relative(int rows)	Works as specified in Java API.
rowDeleted()	Works as specified in Java API.
rowInserted()	Works as specified in Java API.
rowUpdated()	Works as specified in Java API.
setFetchDirection(int direction)	Works as specified in Java API.
setFetchSize(int rows)	No operation in SOLID. Sets the value for the number of rows to be fetched from the database each time. The value a user sets with this method is ignored.
updateAsciiStream(int columnIndex, InputStream x, int length)	Works as specified in Java API.
updateAsciiStream(String columnName, InputStream x, int length)	Works as specified in Java API.
updateBigDecimal(int columnIndex, BigDecimal x)	Works as specified in Java API.
updateBigDecimal(String columnName, BigDecimal x)	Works as specified in Java API.
updateBinaryStream(int columnIndex, InputStream x, int length)	Works as specified in Java API.
updateBinaryStream(String columnName, InputStream x, int length)	Works as specified in Java API.
updateBoolean(int columnIndex, boolean x)	Works as specified in Java API.
updateBoolean(String columnName, boolean x)	Works as specified in Java API.
updateByte(int columnIndex, byte x)	Works as specified in Java API.
updateByte(String columnName, byte x)	Works as specified in Java API.
updateBytes(int columnIndex, byte[] x)	Works as specified in Java API.

updateBytes(String columnName, byte[] x)	Works as specified in Java API.
updateCharacterStream(int columnIndex, Reader x, int length)	Works as specified in Java API.
updateCharacterStream(String columnName, Reader reader, int length)	Works as specified in Java API.
updateDate(int columnIndex, Date x)	Works as specified in Java API.
updateDate(String columnName, Date x)	Works as specified in Java API.
updateDouble(int columnIndex, double x)	Works as specified in Java API.
updateDouble(String columnName, double x)	Works as specified in Java API.
updateFloat(int columnIndex, float x)	Works as specified in Java API.
updateFloat(String columnName, float x)	Works as specified in Java API.
updateInt(int columnIndex, int x)	Works as specified in Java API.
updateInt(String columnName, int x)	Works as specified in Java API.
updateLong(int columnIndex, long x)	Works as specified in Java API.
updateLong(String columnName, long x)	Works as specified in Java API.
updateNull(int columnIndex)	Works as specified in Java API.
updateNull(String columnName)	Works as specified in Java API.
updateObject(int columnIndex, Object x)	Works as specified in Java API.
updateObject(int columnIndex, Object x, int scale)	Works as specified in Java API.
update Object(String columnName, Object x)	Works as specified in Java API.
updateObject(String columnName, Object x, int scale)	Works as specified in Java API.
updateRow()	Works as specified in Java API.
updateShort(int columnIndex, short x)	Works as specified in Java API.
updateShort(String columnName, short x)	Works as specified in Java API.
updateString(int columnIndex, String x)	Works as specified in Java API.
updateString(String columnName, String x)	Works as specified in Java API.
updateTime(int columnIndex, Time x)	Works as specified in Java API.
updateTime(String columnName, Time x)	Works as specified in Java API.

updateTimestamp(int columnIndex, Timestamp x)	Works as specified in Java API.
updateTimestamp(String columnName, Timestamp x)	Works as specified in Java API.
wasNull()	Works as specified in Java API.

ResultSetMetaData

The java.sql.ResultSetMetaData interface is a public abstract interface. This interface is used to find out about the types and properties of the columns in a ResultSet.

SQLData

The java.sql.SQLData interface is not supported. This interface is used to custom map SQL user-defined types. It reflects a SQL3 standard that is currently unavailable in the Solid database.

SQLInput

The java.sql.SQLInput interface is not supported. This interface is an input stream that represents an instance of a SQL structured or distinct type. It reflects a SQL3 standard that is currently unavailable in the SOLID database.

SQLOutput

The java.sql.SQLOutput interface is not supported. This interface is an output stream used to write the attributes of a user-defined type back to the database. It reflects a SQL3 standard that is currently unavailable in the SOLID database.

Statement

The java.sql.Statement interface is a public abstract interface. It is the object used to execute a static SQL statement and obtain the results of the execution.

Subinterfaces:

CallableStatement, PreparedStatement

Methods

Note that SOLID does not support the batch update feature, which allows an application to submit multiple update statements (insert/update/delete) in a single request to the database.

Method name	Notes
addBatch(String sql)	Not supported by SOLID.
cancel()	Works as specified in Java API.
clearBatch()	Not supported by SOLID.
clearWarnings()	Works as specified in Java API.
close()	Works as specified in Java API.
execute(String sql)	Works as specified in Java API.
executeBatch ()	Not supported by SOLID.
executeQuery(String sql)	Works as supported by Java API.
executeUpdate(String sql)	Works as specified in Java API.
getConnection()	Works as specified in Java API.
getFetchDirection()	Works as specified in Java API.
getFetchSize()	No operation in SOLID. The set value a user sets with this method (which is ignored) is returned.
getMaxFieldSize()	Maxfield size does not affect the Solid server's behavior.
getMaxRows()	Works as specified in Java API.
getMoreResults()	Solid does not support multiple resultsets.
getQueryTimeout()	Works as specified in Java API.
getResultSet()	Works as specified in Java API.
getResultSetConcurrency()	Not supported by SOLID.
getResultSetType()	Not supported by SOLID.
getUpdateCount()	Works as specified in Java API.
getWarnings()	Works as specified in Java API.
setCursorName(String name)	Works as specified in Java API.
setEscapeProcessing(boolean enable)	Works as specified in Java API.
setFetchDirection(int direction)	Works as specified in Java API.

setFetchSize(int rows)	No operation in SOLID. Sets the value for the number of rows to be fetched from the database each time. The value a user sets with this method is ignored.
setMaxFieldSize(int max)	Maxfield size does not affect the Solid server's behavior.
setMaxRows(int)	Works as specified in Java API.
setQueryTimeout(int)	Works as specified in Java API.

Struct

The java.sql.Struct interface is not supported. This interface represents the standard mapping in the Java programming language for a SQL structured type. It reflects a SQL3 standard that is currently unavailable in the SOLID database.

ResultSet (updatable)

The java.sql.Resultset interface contains methods for producing `ResultSet` objects that are updatable. A result set is updatable if its concurrency type is CONCUR_UPDATABLE. Rows in an updatable result set may be updated, inserted, and deleted.

Methods

Method name	Notes
updateAsciiStream(int columnIndex, InputStream x, int length)	Works as specified in Java API.
updateAsciiStream(String columnIndex, InputStream x, int length)	Works as specified in Java API.
updateBigDecimal(int columnIndex, BigDecimal x)	Works as specified in Java API.
updateBigDecimal(String columnName, BigDecimal x)	Works as specified in Java API.
updateBinaryStream(int columnIndex, InputStream x, int length)	Works as specified in Java API.
updateBinaryStream(String columnName, InputStream x, int length)	Works as specified in Java API.
updateBoolean(int columnIndex, boolean x)	Works as specified in Java API.

updateBoolean(String columnName, boolean x)	Works as specified in Java API.
updateByte(int columnIndex, byte x)	Works as specified in Java API.
updateByte(String columnName, byte x)	Works as specified in Java API.
updateBytes(int columnIndex, byte[] x)	Works as specified in Java API.
updateBytes(String columnName, byte[] x)	Works as specified in Java API.
updateCharacterStream(int columnIndex, Reader x, int length)	Works as specified in Java API.
updateCharacterStream(String columnName, Reader reader, int length)	Works as specified in Java API.
updateDate(int columnIndex, Date x)	Works as specified in Java API.
updateDate(String columnName, Date x)	Works as specified in Java API.
updateDouble(int columnIndex, double x)	Works as specified in Java API.
updateDouble(String columnName, double x)	Works as specified in Java API.
updateFloat(int columnIndex, float x)	Works as specified in Java API.
updateFloat(String columnName, float x)	Works as specified in Java API.
updateInt(int columnIndex, int x)	Works as specified in Java API.
updateInt(String columnName, int x)	Works as specified in Java API.
updateLong(int columnIndex, long x)	Works as specified in Java API.
updateLong(String columnName, long x)	Works as specified in Java API.
updateNull(int columnIndex)	Works as specified in Java API.
updateNull(String columnName)	Works as specified in Java API.
updateObject(int columnIndex, Object x)	Works as specified in Java API.
updateObject(int columnIndex, Object x, int scale)	Works as specified in Java API.
updateObject(String columnName, Object x)	Works as specified in Java API.
updateObject(String columnName, Object x. int scale)	Works as specified in Java API.
updateRow()	Works as specified in Java API.
updateShort(int columnIndex, short x)	Works as specified in Java API.
updateShort(String columnName, short x)	Works as specified in Java API.

updateString(int columnIndex, String x)	Works as specified in Java API.
updateString(String columnName, String x)	Works as specified in Java API.
updateTime(int columnIndex, Time x)	Works as specified in Java API.
updateTime(String columnName, Time x)	Works as specified in Java API.
updateTimestamp(int columnIndex, Timestamp x)	Works as specified in Java API.
updateTimestamp(String columnName, Timestamp x)	Works as specified in Java API.

Code Examples

Sample 1:

```
/**
 *      sample1 JDBC sample application
 *
 *
 *      This simple JDBC application does the following using
 *      SOLID native JDBC driver.
 *
 *  1. Registers the driver using JDBC driver manager services
 *  2. Prompts the user for a valid JDBC connect string
 *  3. Connects to SOLID using the driver
 *  4. Creates a statement for one query,
 *     'SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM TABLES'
 *     for reading data from one of SOLID system
       tables.
 *  5. Executes the query
 *  6. Fetches and dumps all the rows of a result set.
 *  7. Closes connection
 *
 *  To build and run the application
 *
 *  1. Make sure you have a working Java Development environment
```

```
 *  2. Install and start SOLID to connect. Ensure that the
 *     server is up and running.
 *  3. Append SolidDriver.zip into the CLASSPATH definition used
 *     by your development/running environment.
 *  4. Create a java project based on the file sample1.java.
 *  5. Build and run the application.
 *
 *  For more information read the readme.htm file contained by
 *  SOLID JDBC Driver package.
 *
 */

import java.io.*;

public class sample1 {

    public static void main (String args[]) throws Exception
    {
        java.sql.Connection conn;
        java.sql.ResultSetMetaData meta;
        java.sql.Statement stmt;
        java.sql.ResultSet result;
        int i;

        System.out.println("JDBC sample application starts...");
        System.out.println("Application tries to register the driver.");

        // this is the recommended way for registering Drivers
        java.sql.Driver d =
(java.sql.Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();

        System.out.println("Driver succesfully registered.");
```

```
        // the user is asked for a connect string
        System.out.println("Now sample application needs a connectstring
in format:\n");
        System.out.println("jdbc:solid://<host>:<port>/<user name>/
<password>\n");
        System.out.print("\nPlease enter the connect string >");
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
        String sCon = reader.readLine();

        // next, the connection is attempted
        System.out.println("Attempting to connect :" + sCon);
        conn = java.sql.DriverManager.getConnection(sCon);

        System.out.println("SolidDriver succesfully connected.");

        String sQuery = "SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM
TABLES";

        stmt= conn.createStatement();

        result = stmt.executeQuery(sQuery);
        System.out.println("Query executed and result set obtained.");

        // we get a metadataobject containing information about the
        // obtained result set
        System.out.println("Obtaining metadata information.");
        meta = result.getMetaData();
        int cols = meta.getColumnCount();

        System.out.println("Metadata information for columns is as
follows:");
        // we dump the column information about the result set
        for (i=1; i <= cols; i++)
```

```
        {
            System.out.println("Column i:"+i+"  "+meta.getColumnName(i)+
"," + meta.getColumnType(i) + "," + meta.getColumnTypeName(i));
        }

        // and finally, we dump the result set
        System.out.println("Starting to dump resultset.");
        int cnt = 1;
        while(result.next())
        {
            System.out.print("\nRow "+cnt+" : ");
            for (i=1; i <= cols; i++) {
                System.out.print(result.getString(i)+"\t");
            }
            cnt++;
        }

        stmt.close();

        conn.close();
        // and not it is all over
        System.out.println("\nResult set dumped. Sample application
finishes.");
    }
}
```

Sample 1 output

```
K:\projects\jdbc\prod10\samples>java sample1
JDBC sample application starts...
Application tries to register the driver.
Driver succesfully registered.
Now sample application needs a connectstring in format:

jdbc:solid://<host>:<port>/<user name>/<password>
```

```
Please enter the connect string >jdbc:solid://localhost:1313/dba/dba
Attempting to connect :jdbc:solid://localhost:1313/dba/dba
SolidDriver succesfully connected.
Query executed and result set obtained.
Obtaining metadata information.
Metadata information for columns is as follows:
Column i:1  TABLE_SCHEMA,12,VARCHAR
Column i:2  TABLE_NAME,12,VARCHAR
Column i:3  TABLE_TYPE,12,VARCHAR
Starting to dump resultset.

Row 1 : _SYSTEM SYS_TABLES      BASE TABLE
Row 2 : _SYSTEM SYS_COLUMNS     BASE TABLE
Row 3 : _SYSTEM SYS_USERS       BASE TABLE
Row 4 : _SYSTEM SYS_UROLE       BASE TABLE
Row 5 : _SYSTEM SYS_RELAUTH     BASE TABLE
Row 6 : _SYSTEM SYS_ATTAUTH     BASE TABLE
Row 7 : _SYSTEM SYS_VIEWS       BASE TABLE
Row 8 : _SYSTEM SYS_KEYPARTS    BASE TABLE
Row 9 : _SYSTEM SYS_KEYS        BASE TABLE
Row 10 : _SYSTEM        SYS_CARDINAL    BASE TABLE
Row 11 : _SYSTEM        SYS_INFO        BASE TABLE
Row 12 : _SYSTEM        SYS_SYNONYM     BASE TABLE
Row 13 : _SYSTEM        TABLES  VIEW
Row 14 : _SYSTEM        COLUMNS VIEW
Row 15 : _SYSTEM        SQL_LANGUAGES   BASE TABLE
Row 16 : _SYSTEM        SERVER_INFO     VIEW
Row 17 : _SYSTEM        SYS_TYPES       BASE TABLE
Row 18 : _SYSTEM        SYS_FORKEYS     BASE TABLE
Row 19 : _SYSTEM        SYS_FORKEYPARTS BASE TABLE
Row 20 : _SYSTEM        SYS_PROCEDURES  BASE TABLE
Row 21 : _SYSTEM        SYS_TABLEMODES  BASE TABLE
```

```
Row 22 : _SYSTEM        SYS_EVENTS      BASE TABLE
Row 23 : _SYSTEM        SYS_SEQUENCES   BASE TABLE
Row 24 : _SYSTEM        SYS_TMP_HOTSTANDBY      BASE TABLE
Result set dumped. Sample application finishes.
```

Sample 2

```
/**
 *      sample2 JDBC sample applet
 *
 *
 *      This simple JDBC applet does the following using
 *      Solid native JDBC driver.
 *
 *  1. Registers the driver using JDBC driver manager services
 *  2. Connects to SOLID using the driver.
 *     Used url is read from sample2.html
 *  3. Executes given SQL statements
 *
 *  To build and run the application
 *
 *  1. Make sure you have a working Java Development environment
 *  2. Install and start SOLID to connect. Ensure that
 *      the server is up and running.
 *  3. Append SolidDriver.zip into the CLASSPATH definition used
 *     by your development/running environment.
 *  4. Create a java project based on the file sample2.java.
 *  5. Build and run the application. Check that sample2.html
 *     defines valid url to your environment.
 *
 *  For more information read the readme.htm file contained by
 *  SOLID JDBC Driver package.
 *
 */
```

```
import java.util.*;
import java.awt.*;
import java.applet.Applet;
import java.net.URL;
import java.sql.*;

public class sample2 extends Applet {
    TextField textField;
    static TextArea textArea;

    String url = null;
    Connection con = null;

    public void init() {
        // a valid value for url could be
        // url = "jdbc:solid://localhost:1313/dba/dba";

        url = getParameter("url");

        textField = new TextField(40);
        textArea = new TextArea(10, 40);
        textArea.setEditable(false);

        Font font = textArea.getFont();
        Font newfont = new Font("Monospaced", font.PLAIN, 12);
        textArea.setFont(newfont);

        // Add Components to the Applet.
        GridBagLayout gridBag = new GridBagLayout();
        setLayout(gridBag);
        GridBagConstraints c = new GridBagConstraints();
        c.gridwidth = GridBagConstraints.REMAINDER;
```

```
        c.fill = GridBagConstraints.HORIZONTAL;
        gridBag.setConstraints(textField, c);
        add(textField);

        c.fill = GridBagConstraints.BOTH;
        c.weightx = 1.0;
        c.weighty = 1.0;
        gridBag.setConstraints(textArea, c);
        add(textArea);

        validate();

        try {
            // Load the SOLID JDBC Driver
            Driver d = (Driver)Class.forName
("solid.jdbc.SolidDriver").newInstance();

            // Attempt to connect to a driver.
            con = DriverManager.getConnection (url);

            // If we were unable to connect, an exception
            // would have been thrown.  So, if we get here,
            // we are successfully connected to the url

            // Check for, and display and warnings generated
            // by the connect.
            checkForWarning (con.getWarnings ());

            // Get the DatabaseMetaData object and display
            // some information about the connection
            DatabaseMetaData dma = con.getMetaData ();
```

```
            textArea.appendText("Connected to " + dma.getURL() + "\n");
            textArea.appendText("Driver       " + dma.getDriverName() +
"\n");
            textArea.appendText("Version      " + dma.getDriverVersion()
+ "\n");
        }
        catch (SQLException ex) {
            printSQLException(ex);
        }
        catch (Exception e) {
            textArea.appendText("Exception:  " + e + "\n");
        }
    }

    public void destroy() {
        if (con != null) {
            try {
                con.close();
            }
            catch (SQLException ex) {
                printSQLException(ex);
            }
            catch (Exception e) {
                textArea.appendText("Exception:  " + e + "\n");
            }
        }
    }

    public boolean action(Event evt, Object arg) {
        if (con != null) {
            String sqlstmt = textField.getText();
            textArea.setText("");
            try {
```

```
            // Create a Statement object so we can submit
            // SQL statements to the driver
            Statement stmt = con.createStatement ();
            // set row limit
            stmt.setMaxRows(50);
            // Submit a query, creating a ResultSet object
            ResultSet rs = stmt.executeQuery (sqlstmt);

            // Display all columns and rows from the result set
            textArea.setVisible(false);
            dispResultSet (stmt,rs);
            textArea.setVisible(true);

            // Close the result set
            rs.close();

            // Close the statement
            stmt.close();
        }
        catch (SQLException ex) {
            printSQLException(ex);
        }
        catch (Exception e) {
            textArea.appendText("Exception:  " + e + "\n");
        }
        textField.selectAll();
    }
    return true;
}

//-------------------------------------------------------------------
// checkForWarning
// Checks for and displays warnings.  Returns true if a warning
```

```
// existed
//-------------------------------------------------------------------

private static boolean checkForWarning (SQLWarning warn)
        throws SQLException
{
    boolean rc = false;

    // If a SQLWarning object was given, display the
    // warning messages.  Note that there could be
    // multiple warnings chained together

    if (warn != null) {
        textArea.appendText("\n*** Warning ***\n");
        rc = true;
        while (warn != null) {
            textArea.appendText("SQLState: " +
                warn.getSQLState () + "\n");
            textArea.appendText("Message:  " +
                warn.getMessage () + "\n");
            textArea.appendText("Vendor:   " +
                warn.getErrorCode () + "\n");
            textArea.appendText("\n");
            warn = warn.getNextWarning ();
        }
    }
    return rc;
}

//-------------------------------------------------------------------
// dispResultSet
// Displays all columns and rows in the given result set
//-------------------------------------------------------------------
```

```
private static void dispResultSet (Statement sta, ResultSet rs)
    throws SQLException
{
    int i;

    // Get the ResultSetMetaData.  This will be used for
    // the column headings
    ResultSetMetaData rsmd = rs.getMetaData ();

    // Get the number of columns in the result set
    int numCols = rsmd.getColumnCount ();
    if (numCols == 0) {
        textArea.appendText("Updatecount is "+sta.getUpdateCount());
        return;
    }

    // Display column headings
    for (i=1; i<=numCols; i++) {
        if (i > 1) {
            textArea.appendText("\t");
        }
        try {
            textArea.appendText(rsmd.getColumnLabel(i));
        }
        catch(NullPointerException ex) {
            textArea.appendText("null");
        }
    }
    textArea.appendText("\n");

    // Display data, fetching until end of the result set
    boolean more = rs.next ();
```

```
            while (more) {

                // Loop through each column, get the
                // column datza and display it
                for (i=1; i<=numCols; i++) {
                    if (i > 1) {
                        textArea.appendText("\t");
                    }
                    try {
                        textArea.appendText(rs.getString(i));
                    }
                    catch(NullPointerException ex) {
                        textArea.appendText("null");
                    }
                }
                textArea.appendText("\n");

                // Fetch the next result set row
                more = rs.next ();
            }
        }

        private static void printSQLException(SQLException ex)
        {
                // A SQLException was generated.  Catch it and
                // display the error information.  Note that there
                // could be multiple error objects chained
                // together

                textArea.appendText("\n*** SQLException caught ***\n");

                while (ex != null) {
                    textArea.appendText("SQLState: " +
```

```
                    ex.getSQLState () + "\n");
                textArea.appendText("Message:  " +
                    ex.getMessage () + "\n");
                textArea.appendText("Vendor:   " +
                    ex.getErrorCode () + "\n");
                textArea.appendText("\n");
                ex = ex.getNextException ();
            }
    }
}
```

Sample 3

```
/**
 *      sample3 JDBC sample application
 *
 *
 *      This simple JDBC application does the following using
 *      SOLID native JDBC driver.
 *
 *  1. Registers the driver using JDBC driver manager services
 *  2. Prompts the user for a valid JDBC connect string
 *  3. Connects to SOLID using the driver
 *  4. Drops and creates a procedure sample3. If the procedure
 *     does not exist dumps the related exception.
 *  5. Calls that procedure using java.sql.Statement
 *  6. Fetches and dumps all the rows of a result set.
 *  7. Closes connection
 *
 *  To build and run the application
 *
 *  1. Make sure you have a working Java Development environment
 *  2. Install and start SOLID to connect. Ensure that the
 *     server is up and running.
```

```
 *  3. Append SolidDriver.zip into the CLASSPATH definition used
 *     by your development/running environment.
 *  4. Create a java project based on the file sample3.java.
 *  5. Build and run the application.
 *
 *  For more information read the readme.htm file contained by
 *  SOLID JDBC Driver package.
 *
 */

import java.io.*;
import java.sql.*;

public class sample3 {

    static Connection conn;
    public static void main (String args[]) throws Exception
    {
        System.out.println("JDBC sample application starts...");
        System.out.println("Application tries to register the driver.");

        // this is the recommended way for registering Drivers
        Driver d =
(Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();

        System.out.println("Driver succesfully registered.");

        // the user is asked for a connect string
        System.out.println("Now sample application needs a connectstring
in format:\n");
        System.out.println("jdbc:solid://<host>:<port>/<user name>/
<password>\n");
        System.out.print("\nPlease enter the connect string >");
        BufferedReader reader = new BufferedReader(new
```

```
InputStreamReader(System.in));
        String sCon = reader.readLine();

        // next, the connection is attempted
        System.out.println("Attempting to connect :" + sCon);
        conn = DriverManager.getConnection(sCon);

        System.out.println("SolidDriver succesfully connected.");

        DoIt();

        conn.close();
        // and now it is all over
        System.out.println("\nResult set dumped. Sample application
finishes.");
    }


    static void DoIt() {
        try {
            createprocs();
            PreparedStatement pstmt = conn.prepareStatement("call
sample3(?)");
            // set parameter value
            pstmt.setInt(1,10);

            ResultSet rs = pstmt.executeQuery();
            if (rs != null) {
                ResultSetMetaData md = rs.getMetaData();
                int cols = md.getColumnCount();
                int row = 0;
                while (rs.next()) {
                    row++;
```

```
                String ret = "row "+row+": ";
                for (int i=1;i<=cols;i++) {
                    ret = ret + rs.getString(i) + " ";
                }
                System.out.println(ret);
            }
        }
        conn.commit();
    }
    catch (SQLException ex) {
        printexp(ex);
    }
    catch (java.lang.Exception ex) {
        ex.printStackTrace ();
    }

}

static void createprocs() {
    Statement stmt = null;
    String proc = "create procedure sample3 (limit integer)" +
                  "returns (c1 integer, c2 integer) " +
                  "begin " +
                  "  c1 := 0;" +
                  "  while c1 < limit loop " +
                  "    c2 := 5 * c1;" +
                  "    return row;" +
                  "    c1 := c1 + 1;" +
                  "  end loop;" +
                  "end";

    try {
        stmt = conn.createStatement();
```

```
            stmt.execute("drop procedure sample3");
        } catch (SQLException ex) {
            printexp(ex);
        }

        try {
            stmt.execute(proc);
        } catch (SQLException ex) {
            printexp(ex);
            System.exit(-1);
        }
    }

    public static void printexp(SQLException ex) {
        System.out.println("\n*** SQLException caught ***");
        while (ex != null) {
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("Message:  " + ex.getMessage());
            System.out.println("Vendor:   " + ex.getErrorCode());
            ex = ex.getNextException ();
        }
    }

}
```

Sample 4

```
/**
 *      sample4 JDBC sample application
 *
 *
 *      This simple JDBC application does the following using
 *      SOLID native JDBC driver.
 *
```

```
 *  1. Registers the driver using JDBC driver manager services
 *  2. Prompts the user for a valid JDBC connect string
 *  3. Connects to SOLID using the driver
 *  4. Drops and creates a table sample4. If the table
 *     does not exist dumps the related exception.
 *  5. Inserts file given as an argument to database (method Store)
 *  6. Reads this 'blob' back to file out.tmp (method Restore)
 *  7. Closes connection
 *
 *  To build and run the application
 *
 *  1. Make sure you have a working Java Development environment
 *  2. Install and start SOLID to connect. Ensure that
 *      the server is up and running.
 *  3. Append SolidDriver.zip into the CLASSPATH definition used
 *     by your development/running environment.
 *  4. Create a java project based on the file sample4.java.
 *  5. Build and run the application.
 *
 *  For more information read the readme.htm file contained by
 *  SOLID JDBC Driver package.
 *
 */

import java.io.*;
import java.sql.*;

public class sample4 {

    static Connection conn;
    public static void main (String args[]) throws Exception
    {
        String filename = null;
```

```
        String tmpfilename = null;

        if (args.length < 1) {
            System.out.println("usage: java sample4 <infile>");
            System.exit(0);
        }
        filename = args[0];
        tmpfilename = "out.tmp";
        System.out.println("JDBC sample application starts...");
        System.out.println("Application tries to register the driver.");

        // this is the recommended way for registering Drivers
        Driver d =
(Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();

        System.out.println("Driver succesfully registered.");

        // the user is asked for a connect string
        System.out.println("Now sample application needs a connectstring
in format:\n");
        System.out.println("jdbc:solid://<host>:<port>/<user name>/
<password>\n");
        System.out.print("\nPlease enter the connect string >");
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
        String sCon = reader.readLine();

        // next, the connection is attempted
        System.out.println("Attempting to connect :" + sCon);
        conn = DriverManager.getConnection(sCon);

        System.out.println("SolidDriver succesfully connected.");

        // drop and create table sample4
```

```
        createsample4();
        // insert data into it
        Store(filename);
        // and restore it
        Restore(tmpfilename);

        conn.close();
        // and it is all over
        System.out.println("\nSample application finishes.");
    }


    static void Store(String filename) {
        String sql = "insert into sample4 values(?,?)";
        FileInputStream inFileStream ;
        try {
            File f1 = new File(filename);
            int blobsize = (int)f1.length();
            System.out.println("Inputfile size is "+blobsize);
            inFileStream = new FileInputStream(f1);

            PreparedStatement stmt = conn.prepareStatement(sql);
            stmt.setLong(1, System.currentTimeMillis());
            stmt.setBinaryStream(2, inFileStream, blobsize);
            int rows = stmt.executeUpdate();
            stmt.close();
            System.out.println(""+rows+" inserted.");
            conn.commit();
        }
        catch (SQLException ex) {
            printexp(ex);
        }
        catch (java.lang.Exception ex) {
```

```
            ex.printStackTrace ();
        }

    }

    static void Restore(String filename) {
        String sql = "select id,blob from sample4";
        FileOutputStream outFileStream ;
        try {
            File f1 = new File(filename);
            outFileStream = new FileOutputStream(f1);

            PreparedStatement stmt = conn.prepareStatement(sql);
            ResultSet rs = stmt.executeQuery();
            int readsize = 0;
            while (rs.next()) {
                InputStream in = rs.getBinaryStream(2);
                byte bytes[] = new byte[8*1024];
                int nRead = in.read(bytes);
                while (nRead != -1) {
                    readsize = readsize + nRead;
                    outFileStream.write(bytes,0,nRead);
                    nRead = in.read(bytes);
                }

            }
            stmt.close();
            System.out.println("Read "+readsize+" bytes from database");
        }
        catch (SQLException ex) {
            printexp(ex);
        }
        catch (java.lang.Exception ex) {
```

```
            ex.printStackTrace ();
        }

    }


    static void createsample4() {
        Statement stmt = null;
        String proc = "create table sample4 (" +
                      "id numeric not null primary key,"+
                      "blob long varbinary)";

        try {
            stmt = conn.createStatement();
            stmt.execute("drop table sample4");
        } catch (SQLException ex) {
            printexp(ex);
        }

        try {
            stmt.execute(proc);
        } catch (SQLException ex) {
            printexp(ex);
            System.exit(-1);
        }
    }

    static void printexp(SQLException ex) {
        System.out.println("\n*** SQLException caught ***");
        while (ex != null) {
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("Message:  " + ex.getMessage());
            System.out.println("Vendor:   " + ex.getErrorCode());
```

```
            ex = ex.getNextException ();
        }
    }

}
```

SOLID *JDBC Driver* Type Conversion Matrix

The following conversion matrix shows how the java data type to SQL data type conversion is supported by SOLID *JDBC Driver*. Note that this matrix applies to both ResultSet.getXXX and ResultSet.setXXX methods for getting and setting data. An X indicates that the method is supported by SOLID *JDBC Driver*.

SQL Data Type

Java Data Type (applies to getting and setting data)	TINYINT	SMALLINT	INTEGER	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	CHAR	VARCHAR	LONGVARCHAR	WCHAR	WVARCHAR	LONGWVARCHAR	BINARY	VARBINARY	LONGVARBINARY	*DATE	*TIME	*TIMESTAMP
getArray/setArray																				
getBlob/setBlob																				
getByte/setByte	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
getCharacter-Stream/setCharacterStream									X	X	X	X	X	X	X	X	X	X	X	X
getClob/setClob																				
getShort/setShort	X	X	X	X	X	X	X	X	X	X	X									
getInt/setInt	X	X	X	X	X	X	X	X	X	X	X									
getlong/setLong	X	X	X	X	X	X	X	X	X	X	X									
getfloat/setfloat	X	X	X	X	X	X	X	X	X	X	X									
getDouble/setDouble	X	X	X	X	X	X	X	X	X	X	X									
getBigDecimal/set-BigDecimal	X	X	X	X	X	X	X	X	X	X	X									
getRef/setRef																				
getBoolean/set-Boolean	X	X	X	X	X	X	X	X	X	X	X									
getString/setString	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
getBytes/setBytes									X	X	X	X	X	X	X	X	X			
getDate/setDate									X	X	X	X	X	X				X		X
getTime/setTime									X	X	X	X	X	X					X	X
getTimestamp/set-Timestamp									X	X	X	X	X	X				X		X
getAsciiStream/setAsciiStream									X	X	X	X	X	X	X	X	X			
getUnicodeStream/setUnicodeStream									X	X	X	X	X	X	X	X	X			
getBinaryStream/setBinaryStream									X	X	X	X	X	X	X	X	X			
getObject/setObject	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

A

SOLID Supported ODBC Functions

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
Connecting to a Data Source			
SQLAllocEnv (1.0)	N/A	Deprecated (replaced by **SQLAllocHandle**)	N/A
SQLAllocConnect (1.0)	N/A	Deprecated (replaced by **SQLAllocHandle**)	N/A
SQLAllocHandle (3.0)	Returns the list of supported data source attributes.	Supported	ISO 92
	Returns the list of installed drivers and their attributes.	Supported	ODBC
SQLConnect (1.0)	Establishes connections to a driver and a data source. The connection handle references storage of all information about the connection to the data source, including status, transaction state, and error information.	Supported	ISO 92
SQLDriverConnect (1.0)	This function applies only to Windows environments and is an alternative to **SQLConnect**. It supports data sources that require more connection information than the three arguments in **SQLConnect**, including dialog boxes to prompt the user for all connection information, and data sources that are not defined in the system information.	Supported	ODBC

* Version introduced is the version when the function was initially added to the ODBC API.

** Conformance level can be **ISO 92** (also appears in X/Open version 1 because X/Open is a pure superset of ISO 92), **X/Open** (also appears in ODBC 3.x because ODBC 3.x is a pure superset of X/Open version 1), **ODBC** (appears in neither ISO 92 or X/Open) or **N/A** (Deprecated in ODBC 3.x).

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLBrowseConnect (1.0)	Returns successive levels of attributes and attribute values. When all levels have been enumerated, a connection to the data source is completed and a complete connection string is returned. A return of SQL_SUCCESS_WITH_INFO indicates that all connection information has been specified and the application is now connected to the data source.	Supported	ISO 92
SQLGetInfo (1.0)	Returns general information about the driver and data source associated with a connection.	Supported	ISO 92
SQLGetFunctions (1.0)	Returns information about whether a driver supports a specific ODBC function.	Supported; this function is implemented in the ODBC Driver Manager. It can also be implemented in drivers. If a driver implements **SQLGetFunctions**, the Driver manager calls the function in the driver. Otherwise, it executes the function itself. In Solid's case, the function is implemented in the driver so that the application linked to the driver can also call this function from the application.	ISO 92
SQLGetTypeInfo (1.0)	Returns information about data types supported by the data source. The driver returns the information in the form of a SQL result set. The data types are intended for use in Data Definition Language (DDL) statements.	Supported	ISO 92
Obtaining Information about a Driver and Data Source			
SQLDataSources (1.0)	Returns information about a data source.	Supported; this function is implemented in the ODBC Driver Manager. For non-Microsoft Windows platforms which do not have the Microsoft ODBC Driver manager, this function is not supported.	ISO 92

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLDrivers (2.0)	Lists driver descriptions and driver attribute keywords.	Supported; this function is implemented in the ODBC Driver Manager. For non-Microsoft Windows platforms which do not have the Microsoft ODBC Driver manager, this function is not supported.	ODBC
SQLGetConnectAttr (3.0)	Returns the value of a connection attribute.	Supported	ISO 92
SQLSetConnectAttr (3.0)	Sets a connection attribute.	Supported	ISO 92
SQLGetEnvAttr (3.0)	Returns the value of an environment attribute.	Supported	ISO 92
SQLSetEnvAttr (3.0)	Sets an environment attribute.	Supported	ISO 92
SQLGetStmtAttr (3.0)	Returns the value of a statement attribute.	Supported (replaced by **SQLGetStmtAttr**)	ISO 92
SQLSetStmtAttr (3.0)	Sets a statement attribute.	Supported	ISO 92
SQLSetConnectOption (1.0)	N/A	Deprecated (replaced by **SQLSetConnectAttr**)	N/A
SQLGetConnectOption (1.0)		Deprecated (replaced by **SQLGetConnectAttr**)	N/A
SQLGetStmtOption (1.0)	N/A	Deprecated (replaced by **SQLGetStmtAttr**)	N/A
SQLSetStmtOption (1.0)	N/A	Deprecated (replaced by **SQLSetStmtAttr**)	N/A
Setting and Retrieving Descriptor Fields			
SQLGetDescField (3.0)	Returns the current setting or value of a single descriptor field.	Supported	ISO 92
SQLSetDescField (3.0)	Sets the value of a single field of a descriptor record.	Supported	ISO 92

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLGetDescRec (3.0)	Returns the current settings or values of multiple fields of a descriptor record. The fields returned describe the name, data type, and storage column or parameter data.	Supported	ISO 92
SQLSetDescRec (3.0)	Sets multiple descriptor fields that affect the data type and buffer bound to a column or parameter data.	Supported	ISO 92
SQLCopyDesc (3.0)	Copies descriptor information from one descriptor handle to another.	Supported	ISO 92
Preparing SQL Requests			
SQLAllocStmt (1.0)	N/A	Deprecated (replaced by **SQLAllocHandle**)	N/A
SQLPrepare (1.0)	Prepares a SQL statement for later execution.	Supported	ISO 92
SQLBindParameter (2.0)	Assigns storage for a parameter in a SQL statement.	Supported Note: This function replaces **SQLBindParam** which did not exist in ODBC 2.x, although it is in the X/Open and ISO standards.	ODBC
SQLGetCursorName (1.0)	Returns the cursor name associated with a statement handle.	Supported	ISO 92
SQLSetCursorName (1.0)	Specifies a cursor name with an active statement. If an application does not call **SQLSetCursorName**, the driver generates cursor names as needed for SQL statement processing.	Supported	ISO 92
SQLParamOptions (1.0)	N/A	Deprecated (replaced by **SQLSetStmtAttr**)	N/A
SQLSetParam (1.0)	N/A	Deprecated (replaced by **SQLBindParameter**)	N/A
SQLSetScrollOptions (1.0)	Sets options that control cursor behavior.	Deprecated (replaced by **SQLGetInfo** and **SQLSetStmtAttr**)	ODBC

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
Submitting Requests			
SQLExecute (1.0)	Executes a prepared statement using the current values of the parameter marker variables if any parameter markers exist in the statement.	Supported	ISO 92
SQLExecDirect (1.0)	Executes a preparable statement using the current values of the parameter marker variables if any parameters exist in the statement. **SQLExecDirect i**s the fastest way to submit a SQL statement for one-time execution.	Supported	ISO 92
SQLNativeSQL (1.0)	Returns the SQL string as modified by the driver. **SQLNativeSQL** does not execute the SQL statement.	Not implemented; Solid does not support this functionality.	N/A
SQLDescribeParam (1.0)	Returns the text of a SQL statement as translated by the driver. This information is also available in the fields of the IPD.	Supported	ODBC
SQLNumParams (1.0)	Returns the number of parameters in a SQL statement.	Supported	ISO 92
SQLParamData (1.0)	Used in conjunction with **SQLPutData** to supply parameter data at execution time. (Useful for long data values.)	Supported	ISO 92
SQLPutData (1.0)	Allows an application to send data for a parameter or column to the driver at statement execution time. This function can be used to send character or binary data values in parts to a column with a character, binary, or data source-specific data type (for example, parameters of the SQL_LONGVARBINARY or SQL_LONGVARCHAR types).	Supported	ISO 92
Retrieving Results and Information about Results			
SQLRowCount (1.0)	Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement.	Supported	ISO 92
SQLNumResultCols (1.0)	Returns the number of columns in a result set.	Supported	ISO 92

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLDescribeCol (1.0)	Returns the result descriptor (column name, type, column size, decimal digits, and nullability) for one column in the result set. This information is also available in the fields of the IRD.	Supported	ISO 92
SQLColAttributes (1.0)	N/A	Deprecated (replaced by **SQLColAttribute**)	N/A
SQLColAttribute (3.0)	Describes attributes of a column in the result set.	Supported	ISO 92
SQLBindCol (1.0)	Assigns storage for a result column and specifies the data type.	Supported	ISO 92
SQLFetch (1.0)	Returns multiple result rows, fetching the next rowset of data from the result set and returning data for all bound columns.	Supported	ISO 92
SQLExtendedFetch (1.0)	N/A	Deprecated (replaced by **SQLFetchScroll**)	N/A
SQLFetchScroll (3.0)	Returns scrollable result rows, fetching the specified rowset of data from the result set and returning data for all bound columns. When working with an ODBC 2.x driver, the Driver Manager maps this function to **SQLExtendedFetch**.	Supported Note: Block cursors are not supported. For scrollable cursors, previous and next are supported; however, absolute and relative fetches are not supported.	ISO 92
SQLGetData (1.0)	Returns part or all of one column of one row of a result set. It can be called multiple times to retrieve variable length data in parts, making it useful for long data values.	Supported	ISO 92
SQLSetPos (1.0)	Positions a cursor within a fetched block of data and allows an application to refresh data in the rowset or to update or delete data in the result set.	Not supported	ODBC
SQLBulkOperations (3.0)	Performs bulk insertions and bulk bookmark operations, including update, delete, and fetch by bookmark.	Not supported	ODBC

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLMoreResults (1.0)	Determines whether there are more results available on a statement containing SELECT, UPDATE, INSERT, or DELETE statement and, if so, initializes processing for those results.	Not implemented; SOLID *Embedded Engine* does not support multiple results.	ODBC
SQLGetDiagField (3.0)	Returns additional diagnostic information (a single field of the diagnostic data structure associated with a specified handle). This information includes error, warning, and status information.	Supported	ISO 92
SQLGetDiagRec (3.0)	Returns additional diagnostic information (multiple fields of the diagnostic data structure). Unlike **SQLGetDiagField**, which returns one diagnostic field per call, **SQLGetDiagRec** returns several commonly used fields of a diagnostic record, including the SQLSTATE, the native error code, and the diagnostic message text.	Supported	ISO 92
SQLError (1.0)	N/A	Deprecated (replaced by **SQLGetDiagRec**)	N/A
Obtaining Information about the Data Source's System Tables			
SQLColumnPrivileges (1.0)	Returns a list of columns and associated privileges for the specified table. The driver returns the information as a result set on the specified *StatementHandle.* This function is supported via an appropriate SQL execution.	Supported	ODBC
SQLColumns (1.0)	Returns a list of columns and associated privileges for the specified table. The driver returns the information as a result set on the specified *StatementHandle.* This function is supported via an appropriate SQL execution.	Supported	X/Open

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLForeignKeys (1.0)	Returns two type of lists: ■ Foreign keys in the specified table (columns in the specified table that refer to primary keys in other tables). ■ Foreign keys in other tables that refer to the primary key in the specified table. The driver returns each list as a result set on the specified statement.	Supported	ODBC
SQLPrimaryKeys (1.0)	Returns the list of column names that make up the primary key for a table. The driver returns the information as a result set. This function does not support returning primary keys from multiple tables in a single call.	Supported	ODBC
SQLProcedureColumns (1.0)	Returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures. The driver returns the information as a result set on the specified statement.	Supported.	ODBC
SQLProcedures (1.0)	Returns the list of procedure names stored in a specific data source. *Procedure* is a generic term used to describe an executable object, or a named entity that can be invoked using input and output parameters.	Supported	ODBC
SQLSpecialColumns (1.0)	Returns the following information about columns within a specified table: ■ The optimal set of columns that uniquely identifies a row in the table. ■ Columns that are automatically updated when any value in the row is updated by a transaction.	Supported	X/Open
SQLStatistics (1.0)	Returns statistics about a single table and the list of indexes associated with the table. The driver returns the information as a result set.	Supported	ISO 92
SQLTablePrivileges (1.0)	Returns a list of tables and the privileges associated with each table. The driver returns the information as a result set on the specified statement.	Supported	ODBC

Function Names/Version Introduced*	Purpose	Availability when using ODBC	Conformance**
SQLTables (1.0)	Returns the list of table, catalog, or schema names, and table types, stored in a specific data source.	Supported	X/Open
Terminating a statement			
SQLFreeStmt (1.0)	Ends statement processing, discards pending results, and optionally, frees all resources associated with the statement handle.	Supported Note: The **SQLFreeStmt** with an option of SQL_DROP is replaced by **SQLFreeHandle**.	ISO 92
SQLCloseCursor (3.0)	Closes a cursor that has been opened on a statement, and discards pending results.	Supported	ISO 92
SQLCancel (1.0)	Cancels the processing on a SQL statement.	Supported	ISO 92
SQLEndTran (3.0)	Requests a transaction commit or rollback on all statements associated with a connection. **SQLEndTran** can also request that a commit or rollback operation be performed for all connections associated with an environment.	Supported	ISO 92
SQLTransact (1.0)	N/A	Deprecated (replaced by **SQLEndTran**)	N/A
Terminating a Connection			
SQLDisconnect (1.0)	Closes the connection associated with a specific connection handle.	Supported	ISO 92
SQLFreeConnect (1.0)	N/A	Deprecated (replaced by **SQLFreeHandle**)	N/A
SQLFreeEnv (1.0)	N/A	Deprecated (replaced by **SQLFreeHandle**)	N/A
SQLFreeHandle (3.0)	Frees resources associated with a specific environment, environment, connection, statement, or descriptor handle	Supported	ISO 92

B

Error Codes

This appendix contains an Error Codes Table that provides possible SQLSTATE values that a driver returns for the **SQLGetDiagRec** function. Note that **SQLGetDiagRec** and **SQLGetDiagField** return SQLSTATE values that conform to the X/Open Data Management: Structured Query Language (SQL), Version 2 (3/95).

Error Codes Table Convention

SQLSTATE values are strings that contain five characters; the first two is a string class value, followed by a three-character subclass value. For example **01000** has **01** as its class value and **000** as its subclass value. Note that a subclass value of 000 means there is no subclass for that SQLSTATE. Class and subclass values are defined in SQL-92.

Class value	Meaning
01	Indicates a warning and includes a return code of SQL_SUCCESS_WITH_INFO.
01, 07, 08, 21, 22, 25, 28, 34, 3C, 3D, 3F, 40, 42, 44, HY	Indicates an error that includes a return value of SQL_ERROR.
IM	Indicates warning and errors that are derived from ODBC.

Note

Typically, when a function successfully executes, it returns a value of SQL_SUCCESS; in some cases, however, the function may also return the SQLSTATE 00000, which also indicates successful execution.

.

SQLSTATE	Error	Can be returned from
01000	General warning	All ODBC functions except: SQLGetDiagField SQLGetDiagRec
01001	Cursor operation conflict	SQLExecDirect SQLExecute SQLParamData
01002	Disconnect error	SQLDisconnect
01003	NULL value eliminated in set function	SQLExecDirect SQLExecute SQLParamData
01004	String data, right truncated	SQLBrowseConnect SQLColAttribute SQLDataSources SQLDescribeCol SQLDriverConnect SQLDrivers SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetConnectAttr SQLGetCursorName SQLGetData SQLGetDescField SQLGetDescRec SQLGetEnvAttr SQLGetInfo SQLGetStmtAttr SQLParamData SQLPutData SQLSetCursorName
01006	Privilege not revoked	SQLExecDirect SQLExecute SQLParamData
01007	Privilege not granted	SQLExecDirect SQLExecute SQLParamData

SQLSTATE	Error	Can be returned from
01S00	Invalid connection string attribute	SQLBrowseConnect SQLDriverConnect
01S01	Error in row	SQLExtendedFetch
01S02	Option value changed	SQLBrowseConnect SQLConnect SQLDriverConnect SQLExecDirect SQLExecute SQLParamData SQLPrepare SQLSetConnectAttr SQLSetDescField SQLSetEnvAttr SQLSetStmtAttr
01S06	Attempt to fetch before the result set returned the first rowset	SQLExtendedFetch SQLFetchScroll
01S07	Fractional truncation	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData
01S08	Error saving File DSN	SQCriverConnect
01S09	Invalid keyword	SQLDriverConnect
07001	Wrong number of parameters	SQLExecDirect SQLExecute
07002	COUNT field incorrect	SQLExecDirect SQLExecute SQLParamData
07005	Prepared statement not a *cursor_specification*	SQLColAttribute SQLDescribeCol

SQLSTATE	Error	Can be returned from
07006	Restricted data type attribute violation	SQLBindCol SQLBindParameter SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData
07009	Invalid descriptor index	SQLBindCol SQLBindParameter SQLColAttribute SQLDescribeCol SQLDesribeParam SQLFetch SQLFetchScroll SQLGetData SQLGetDescField SQLParamData SQLSetDescField SQLSetDescRec SetSetPos
07S01	Invalid use of default parameter	SQLExecDirect SQLExecute SQLParamData SQLPutData
08001	Client unable to establish connection	SQLBrowseConnect SQLConnect SQLDriverConnect
08002	Connection name in use	SQLBrowseConnect SQLConnect SQLDriverConnect SQLSetConnectAttr
08003	Connection does not exist	SQLAllocHandle SQLDisconnect SQLEndTran SQLGetConnectAttr SQLGetInfo SQLSetConnectAttr

SQLSTATE	Error	Can be returned from
08004	Server rejected the connection	SQLBrowseConnect SQLConnect SQLDriverConnect
08007	Connection failure during transaction	SQLEndTran
08S01	Communication link failure	SQLBrowseConnect SQLColumnPrivileges SQLColumns SQLConnect SQLCopyDesc SQLDescribeCol SQLDescribeParam SQLDriverConnect SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLForeignKeys SQLGetConnectAttr SQLGetData SQLGetDescField SQLGetDescRec SQLGetFunctions SQLGetInfo SQLGetTypeInfo SQLMoreResults SQLNumParams SQLNumResultCols SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLSetConnectAttr SQLSetDescField SQLSetDescRec

SQLSTATE	Error	Can be returned from
08S01 (continued)	Communication link failure	SQLSetEnvAttr SQLSetStmtAttr SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
21S01	Insert value list does not match column list	SQLExecDirect SQLPrepare
21S02	Degree of derived table does not match column list	SQLExecDirect SQLExecute SQLParamData SQLPrepare
22001	String data, right truncated	SQLExecDirect SQLExecute SQLFetch SQLFetchScroll SQLParamData SQLPutData SQLSetDescField
22002	Indicator variable required but not supplied	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData
22003	Numeric value out of range	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLGetInfo SQLParamData SQLPutData

SQLSTATE	Error	Can be returned from
22007	Invalid datetime format	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData
22008	Datetime field overflow	SQLExecDirect SQLExecute SQLParamData SQLPutData
22012	Division by zero	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData
22015	Interval field overflow	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData
22018	Invalid character value for cast specification	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData
22019	Invalid escape character	SQLExecDirect SQLExecute SQLPrepare

SQLSTATE	Error	Can be returned from
22025	Invalid escape sequence	SQLExecDirect SQLExecute SQLPrepare
22026	String data, length mismatch	SQLParamData
23000	Integrity constraint violation	SQLExecDirect SQLExecute SQLParamData
24000	Invalid cursor state	SQLCloseCursor SQLColumnPrivileges SQLColumns SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLForeignKeys SQLGetData SQLGetStmtAttr SQLGetTypeInfo SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLConnectAttr SQLSetCursorName SQLSetStmtAttr SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
25000	Invalid transaction state	SQLDisconnect
25S01	Transaction state	SQLEndTran
25S02	Transaction is still active	SQLEndTran
25S03	Transaction is rolled back	SQLEndTran
28000	Invalid authorization specification	SQLBrowseConnect SQLConnect SQLDriverConnect

SQLSTATE	Error	Can be returned from
34000	Invalid cursor name	SQLExecDirect SQLPrepare SQLSetCursorName
3C000	Duplicate cursor name	SQLSetCursorName
3D000	Invalid catalog name	SQLExecDirect
3F000	Invalid schema name	SQLExecDirect SQLPrepare
40001	Serialization failure	SQLColumnPrivileges SQLColumns SQLEndTran SQLExecDirect SQLExecute SQLFetch SQLFetchScroll SQLForeignKeys SQLGetTypeInfo SQLMoreResults SQLParamData SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
40002	Integrity constraint violation	SQLEndTran

SQLSTATE	Error	Can be returned from
40003	Statement completion unknown	SQLColumnPrivileges SQLColumns SQLExecDirect SQLExecute SQLFetch SQLFetchScroll SQLForeignKeys SQLGetTypeInfo SQLMoreResults SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLParamData SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
42000	Syntax error or access violation	SQLExecDirect SQLExecute SQLParamData SQLPrepare
42S01	Base table or view already exists	SQLExecDirect SQLPrepare
42S02	Base table or view not found	SQLExecDirect SQLPrepare
42S11	Index already exists	SQLExecDirect SQLPrepare
42S12	Index not found	SQLExecDirect SQLPrepare
42S21	Column already exists	SQLExecDirect SQLPrepare
42S22	Column not found	SQLExecDirect SQLPrepare
44000	WITH CHECK OPTION violation	SQLExecDirect SQLExecute SQLParamData

SQLSTATE	Error	Can be returned from
HY000	General Error	All ODBC functions except: SQLGetDiagField SQLGetDiagRec
HY001	Memory allocation error	All ODBC function except: SQLGetDiagField SQLGetDiagRec
HY003	Invalid application buffer type	SQLBindCol SQLBindParameter SQLGetData
HY004	Invalid SQL data type	SQLBindParameter SQLGetTypeInfo
HY007	Associated statement is not prepared	SQLCopyDesc SQLGetDescField SQLGetDescRec
HY008	Operation canceled	All ODBC functions that can be processed asynchronously: SQLColAttribute SQLColumnPrivileges SQLColumns SQLDescribeCol SQLDescribeParam SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLForeignKeys SQLGetData SQLGetTypeInfo SQLMoreResults SQLNumParams SQLNumResultCols

SQLSTATE	Error	Can be returned from
HY008 (continued)	Operation canceled	SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
HY009	Invalid use of null pointer	SQLAllocHandle SQLBindParameter SQLColumnPrivileges SQLColumns SQLExecDirect SQLForeignKeys SQLGetCursorName SQLGetData SQLGetFunctions SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLSetConnectAttr SQLSetCursorName SQLSetEnvAttr SQLSetStmtAttr SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables

SQLSTATE	Error	Can be returned from
HY010	Function sequence error	SQLAllocHandle SQLBindCol SQLBindParameter SQLCloseCursor SQLColAttribute SQLColumnPrivileges SQLColumns SQLCopyDesc SQLDescribeCol SQLDescribeParam SQLDisconnect SQLEndTran SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll
HY010	Function sequence error	SQLForeignKeys SQLFreeHandle SQLFreeStmt SQLGetConnectAttr SQLGetCursorName SQLGetData SQLGetDescField SQLGetDescRec SQLGetFunctions SQLGetStmtAttr SQLGetTypeInfo SQLMoreResults SQLNumParams SQLNumResultCols SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLRowCount SQLSetConnectAttr SQLSetCursorName SQLSetDescField

SQLSTATE	Error	Can be returned from
HY010 (continued)	Function sequence error	SQLSetEnvAttr SQLSetDescRec SQLSetStmtAttr SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
HY011	Attribute cannot be set now	SQLParamData SQLSetConnectAttr SQLSetStmtAttr
HY012	Invalid transaction operation code	SQLEndTran
HY013	Memory Management err	All ODBC functions except: SQLGetDiagField SQLGetDiagRec
HY014	Limit on the number of handles exceeded	SQLAllocHandle
HY015	No cursor name available	SQLGetCursorName
HY016	Cannot modify an implementation row descriptor	SQLCopyDesc SQLSetDescField SQLSetDescRec
HY017	Invalid use of an automatically allocated descriptor handle	SQLFreeHandle SQLSetStmtAttr
HY018	Server declined cancel request	SQLCancel
HY019	Non-character and non-binary data sent in pieces	SQLPutData
HY020	Attempt to concatenate a null value	SQLPutData
HY021	Inconsistent descriptor information	SQLBindParameter SQLCopyDesc SQLGetDescField SQLSetDescField SQLSetDescRec
HY024	Invalid attribute value	SQLSetConnectAttr SQLSetEnvAttr SQLSetStmtAttr

SQLSTATE	Error	Can be returned from
HY090	Invalid string or buffer length	SQLBindCol SQLBindParameter SQLBrowseConnect SQLColAttribute SQLColumnPrivileges SQLColumns SQLConnect SQLDataSources SQLDescribeCol SQLDriverConnect SQLDrivers SQLExecDirect SQLExecute SQLFetch SQLFetchScroll
IM001	Driver does not support this function	SQLForeignKeys SQLGetConnectAttr SQLGetCursorName SQLGetData SQLGetDescField SQLGetInfo SQLGetStmtAttr SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLSetConnectAttr SQLSetCursorName SQLSetDescField SQLSetDescRec SQLSetEnvAttr SQLSetStmtAttr SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
HY091	Invalid descriptor field identifier	SQLColAttribute SQLGetDescField SQLSetDescField

SQLSTATE	Error	Can be returned from
HY092	Invalid attribute/option identifier	SQLAllocHandle SQLCopyDesc SQLDriverConnect SQLEndTran SQLFreeStmt SQLGetConnectAttr SQLGetEnvAttr SQLGetStmtAttr SQLParamData SQLSetConnectAttr SQLSetDescField SQLSetEnvAttr SQLSetStmtAttr
HY095	Function type out of range	SQLGetFunctions
HY096	Invalid information type	SQLGetInfo
HY097	Column type out of range	SQLSpecial Columns
HY098	Scope type out of range	SQLSpecial Columns
HY099	Nullable type out of range	SQLSpecial Columns
HY100	Uniqueness option type out of range	SQLStatistics
HY101	Accuracy option type out of range	SQLStatistics
HY103	Invalid retrieval code	SQLDataSources SQLDrivers
HY104	Invalid precision or scale value	SQLBindParameter
HY105	Invalid parameter type	SQLBindParameter SQLExecDirect SQLExecute SQLParamData SQLSetDescField
HY106	Fetch type out of range	SQLExtendedFetch SQLFetchScroll
HY107	Row value out of range	SQLExtendedFetch SQLFetch SQLFetchScroll

SQLSTATE	Error	Can be returned from
HY109	Invalid cursor position	SQLExecDirect SQLExecute SQLGetData SQLGetStmtAttr SQLParamData
HY110	Invalid driver completion	SQLDriverConnect
HY111	Invalid bookmark value	SQLExtendedFetch SQLFetchScroll
HYC00	Optional feature not implemented	SQLBindCol SQLBindParameter SQLColAttribute SQLColumnPrivileges SQLColumns SQLDriverConnect SQLEndTran SQLConnect SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLForeignKeys SQLGetConnectAttr SQLGetData SQLGetEnvAttr SQLGetInfo SQLGetStmtAttr SQLGetTypeInfo SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLSetConnectAttr SQLSetEnvAttr SQLSetStmtAttr SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables

SQLSTATE	Error	Can be returned from
HYT00	Timeout expired	SQLBrowseConnect SQLColumnPrivileges SQLColumns SQLConnect SQLDriverConnect SQLExecDirect SQLExecute SQLExtendedFetch SQLForeignKeys SQLGetTypeInfo SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
HYT01	Connection timeout expired	All ODBC functions except: SQLDrivers SQLDataSources SQLGetEnvAttr SQLSetEnvAttr
IM001	Connection timeout expired	All ODBC functions except: SQLDrivers SQLDataSources SQLGetEnvAttr SQLSetEnvAttr
S0002	Base table not found	SQLExecDirect SQLPrepare
S0011	Index already exists	SQLExecDirect SQLPrepare
S0012	Index not found	SQLExecDirect SQLPrepare
S0021	Column already exists	SQLExecDirect SQLPrepare

SQLSTATE	Error	Can be returned from
S0022	Column not found	SQLExecDirect SQLPrepare
S1000	General error	All ODBC functions except: SQLAllocEnv
S1001	Memory allocation failure	All ODBC functions except: SQLAllocEnv SQLFreeConnect SQLFreeEnv
S1002	Invalid column number	SQLBindCol SQLColAttributes SQLDescribeCol SQLExtendedFetch SQLFetch SQLGetData
S1003	Program type out of range	SQLBindCol SQLBindParameter SQLGetData
S1004	SQL data type out of range	SQLBindParameter SQLGetTypeInfo
S1008	Operation canceled	All ODBC functions that can be processed asynchronously: SQLColAttributes SQLColumnPrivileges SQLColumns SQLDescribeCol SQLDescribeParam SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLForeignKeys SQLGetData SQLGetTypeInfo SQLMoreResults SQLNumParams SQLNumResultCols SQLParamData

SQLSTATE	Error	Can be returned from
S1008 (continued)	Operation canceled	SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
S1009	Invalid argument value	SQLAllocConnect SQLAllocStmt SQLBindCol SQLBindParameter SQLExecDirect SQLForeignKeys SQLGetData SQLGetInfo SQLPrepare SQLPutData SQLSetConnectOption SQLSetCursorName SQLSetStmtOption
S1010	Function sequence error	SQLBindCol SQLBindParameter SQLColAttributes SQLColumnPrivileges SQLColumns SQLDescribeCol SQLDescribeParam SQLDisconnect SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLForeignKeys SQLFreeConnect SQLFreeEnv SQLFreeStmt SQLGetConnectOption SQLGetCursorName SQLGetData

SQLSTATE	Error	Can be returned from
S1010 (continued)	Function sequence error	SQLGetFunctions SQLGetStmtOption SQLGetTypeInfo SQLMoreResults SQLNumParams SQLNumResultCols SQLParamData SQLParamOptions SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLRowCount SQLSetConnectOption SQLSetCursorName SQLSetScrollOptions SQLSetStmtOption SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables SQLTransact
S1011	Operation invalid at this time	SQLGetStmtOption SQLSetConnectOption SQLSetStmtOption
S1012	Invalid transaction operation code specified	SQLTransact
S1015	No cursor name available	SQLGetCursorName

SQLSTATE	Error	Can be returned from
S1090	Invalid string or buffer length	SQLBindCol SQLBindParameter SQLBrowseConnect SQLColAttributes SQLColumnPrivileges SQLColumns SQLConnect SQLDataSources SQLDescribeCol SQLDriverConnect SQLDrivers SQLExecDirect SQLExecute SQLForeignKeys SQLGetCursorName SQLGetData SQLGetInfo SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLSetCursorName SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
S1091	Descriptor type out of range	SQLColAttributes
S1092	Option type out of range	SQLFreeStmt SQLGetConnectOption SQLGetStmtOption SQLSetConnectOption SQLSetStmtOption
S1093	Invalid parameter number	SQLBindParameter SQLDescribeParam
S1094	Invalid scale value	SQLBindParameter
S1095	Function type out of range	SQLGetFunctions
S1096	Information type out of range	SQLGetInfo
S1097	Column type out of range	SQLSpecialColumns

SQLSTATE	Error	Can be returned from
S1098	Scope type out of range	SQLSpecialColumns

C

SQL Minimum Grammar

An ODBC driver must support a subset of SQL-92 Entry level syntax. This appendix describes this SQL minimum syntax that an ODBC driver must support. An application that uses this syntax will be supported by any ODBC-compliant driver.

Applications can call **SQLGetInfo** with the SQL_SQL_CONFORMANCE to determine if additional features of SQL-92, not covered in this appendix, are supported.

Note

If the driver supports only read-only data sources, the SQL syntax that applies to changing data may not apply to the driver. Applications need to call **SQLGetInfo** with the SQL_DATA_SOURCE_READ_ONLY information type to determine if a data source is read-only.

SQL Statements

create-table-statement ::=
CREATE TABLE *base_table_name*
(*column_identifier data_type* [, *column_identifier data_type*]...)

Important

As the *data_type* in a *create_table_statement*, applications require a data type from the TYPE_NAME column of the result set returned by **SQLGetTypeInfo**.

delete_statement_searched ::=

DELETE FROM *table_name* [WHERE *search_condition*]

drop_table_statement ::=
DROP TABLE *base_table_name*

select_statement ::=
SELECT [ALL | DISTINCT] *select_list*
FROM *table_reference_list*
[WHERE *search_condition*]
[*order_by_clause*]

statement ::= *create_table_statement* |
delete_statement_searched |
drop_table_statement |
insert_stetement |
select_statement |
update_statement_searched

Update_statement_searched ::=
UPDATE *table_name*
SET *column_identifier* = {*expression* |
NULL}
[, *column_identifier* = {*expression* |
NULL}]...
[WHERE *search_condition*]

SQL Statement Elements

base_table_identifier ::= *user_defined_name*

base_table_name ::= *base_table_identifier*

boolean_factor ::= [NOT] *boolean_primary*

boolean_primary ::= *predicate* | (*search_condition*)

boolean_term ::= *boolean_factor* [AND *boolean_term*]

character_string_literal :: = ''{*character*}...''

(*character* is any character in the character set of the driver/data source. To include a single literal quote character (') in a *character_string_literal*, use two literal quote characters [''''].)

column_identifier ::= *user_defined_name*

column_name ::= [*table_name*.]*column_identifie*r

comparison_operator ::= < | > | <= | >= | = | <>

comparison_predicate ::= *expression comparison_operator expression*

data_type ::= *character_string_type*

(*character_string_type* is any data type for which the ""DATA_TYPE"" column in the result set returned by **SQLGetTypeInfo** is either SQL_CHAR or SQLVARCHAR.)

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

dynamic_parameter ::= ?

expression ::= *term* | *expression* {+|–} *term*

factor ::= [+|–]*primary*

insert_value ::= *dynamic_parameter* | *literal* | NULL | USER

letter ::= *lower_case_letter* | *upper_case_letter*

literal ::= *character_string_literal*

lower_case_letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

order_by_clause ::= ORDER BY *sort_specification* [, *sort_specification*]...

primary ::= *column_name* | *dynamic_parameter* | *literal* | (*expression*)

search_condition ::= *boolean_term* [OR *search_condition*]

select_list ::= * | *select_sublist* [, *select_sublist*]...

(*select_list* cannot contain parameters.)

select_sublist ::= *expression*

sort_specification ::= {*unsigned_integer* | *column_name* } [ASC | DESC]

table_identifier ::= *user_defined_name*

table_name ::= *table_identifier*

table_reference ::= *table_name*

table_reference ::= *table_name* [,*table_reference*]...

term ::= *factor* | *term* {*|/} *factor*

unsigned_integer ::= {*digit*}

upper_case_letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

user_defined_name ::= *letter*[*digit* | *letter*| _]...

Data Type Support

At minimum, ODBC drivers must support either SQL_CHAR or SQL_VARCHAR. Other data types support is determined by the driver's or data source's SQL-92 conformance level. To determine the SQL-92 conformance level for a driver or data source, applications need to call **SQLGetTypeInfo**.

Parameter Data Types

Even though each parameter specified with **SQLBindParameter** is defined using a SQL data type, the parameters in a SQL statement have no intrinsic data type. Therefore, parameter markers can be included in a SQL statement only if their data types can be inferred from another operand in the statement. For example, in an arithmetic expression such as **? + COLUMN1**, the data type of the parameter can be inferred from the data type of the named column represented by COLUMN1. An application cannot use a parameter marker if the data type cannot be determined.

The following table describes how a data type is determined for several types of parameters according to SQL-92 standards. For comprehensive information on inferring the parameter type, see the SQL-92 specification.

Location of Parameter	Assumed Data Type
One operand of a binary arithmetic or comparison operator	Same as the other operand
The first operand in a **BETWEEN** clause	Same as the second operand
The second or third operand in a **BETWEEN** clause	Same as the first operand
An expression used with **IN**	Same as the first value or the result column of the subquery
A value used with **IN**	Same as the expression or the first value if there is a parameter marker in the expression
A pattern value used with **LIKE**	VARCHAR
An update value used with **UPDATE**	Same as the update column

Parameter Markers

According to the SQL-92 specification, an application cannot place parameter markers in the following locations:

- In a **SELECT** list.
- As both *expressions* in a *comparison-predicate*.
- As both operands of a binary operator.
- As both the first and second operands of a **BETWEEN** operation.
- As both the first and third operands of a **BETWEEN** operation.
- As both the expression and the first value of an **IN** operation.
- As the operand of a unary + or – operation.
- As the argument of a *set-function-reference*.

For a comprehensive list and more details, see the SQL-92 specification.

Literals in ODBC

The ODBC literal syntax in this section is provided to aid driver writers who are converting a character string type to a numeric or interval type, or from a numeric or interval type to a character string type.

Interval Literal Syntax

The following syntax is used for interval literals in ODBC.

interval_literal ::= *INTERVAL* [+|_] *interval_string interval_qualifier*

interval_string ::= *quote* { *year_month_literal* | *day_time_literal* } *quote*

year_month_literal ::= *years_value* | [*years_value*] *months_valu*e

day_time_literal ::= *day_time_interval* | *time_interval*

day_time_interval ::= *days_value* [*hours_value* [:*minutes_value*[:*seconds_value*]]]

time_interval ::= *hours_value* [:*minutes_value* [:*seconds_value*]]

| *minutes_value* [:*seconds_value*]

| *seconds_value*

years_value ::= *datetime_value*

months_value ::= *datetime_value*

days_value ::= *datetime_value*

hours_value ::= *datetime_value*

minutes_value ::= *datetime_value*

seconds_value ::= *seconds_integer_value* [.[*seconds_fraction*]]

seconds_integer_value ::= *unsigned_integer*

seconds_fraction ::= *unsigned_integer*

datetime_value ::= *unsigned_integer*

interval_qualifier ::= *start_field* TO *end_field* | *single_datetime_field*

start_field ::= *non_second_datetime_field* [(*interval_leading_field_precision*)]

end_field ::= *non_second_datetime_field*

| SECOND[(*interval_fractional_seconds_precision*)]

single_datetime_field ::= *non_second_datetime_field* [(*interval_leading_field_precision*)] | SECOND[(*interval_leading_field_precision* [, (*interval_fractional_seconds_precision*)]

datetime_field ::= *non_second_datetime_field* | SECOND

non_second_datetime_field ::= YEAR | MONTH | DAY | HOUR | MINUTE

interval_fractional_seconds_precision ::= *unsigned_integer*

interval_leading_field_precision ::= *unsigned_integer*

quote ::= '

u*nsigned_integer* ::= *digit…*

Numeric Literal Syntax

The following syntax is used for numeric literals in ODBC:

numeric_literal ::= *signed_numeric_literal* | *unsigned_numeric_literal*

signed_numeric_literal ::= [*sign*] *unsigned_numeric_literal*

unsigned_numeric_literal ::= *exact_numeric_literal* | *approximate_numeric_literal*

exact_numeric_literal ::= *unsigned_integer* [*period*[*unsigned_integer*]] | *period unsigned_integer*

sign ::= *plus_sign* | *minus_sign*

approximate_numeric_literal ::= *mantissa E exponent*

mantissa ::= *exact_numeric_literal*

exponent ::= *signed_integer*

signed_integer ::= [*sign*] *unsigned_integer*

unsigned_integer ::= *digit...*

plus_sign ::= +

minus_sign ::= _

digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

period ::= .

List of Reserved Keywords

The following words are reserved for use in ODBC function calls. These words do not constrain the minimum SQL grammar; however, to ensure compatibility with drivers that support the core SQL grammar, applications should avoid using any of these keywords. The #**define** value SQL_ODBC_KEYWORDS contains a comma-separated list of these keywords.

For a complete list of reserved keywords in several SQL standards and SOLID *ODBC API*, see *the appendix on Reserved Words* in the **SOLID *Embedded Engine* Administrator Guide or SOLID *SynchroNet* Guide**.

ABSOLUTE	ACTION
ADA	ADD
ALL	ALLOCATE
ALTER	AND
ANY	ARE
AS	ASC
ASSERTION	AT
AUTHORIZATION	AVG
BEGIN	BETWEEN
BIT	BIT_LENGTH

BOTH	BY
CASCADE	CASCADED
CASE	CAST
CATALOG	CHAR
CHAR_LENGTH	CHARACTER
CHARACTER_LENGTH	CHECK
CLOSE	COALESCE
COLLATE	COLLATION
COLUMN	COMMIT
CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS
CONTINUE	CONVERT
CORRESPONDING	COUNT
CREATE	CROSS
CURRENT	CURRENT_DATE
CURRENT_TIME	CURRENT_TIMESTAMP
CURRENT_USER	CURSOR
DATE	DAY
DEALLOCATE	DEC
DECIMAL	DECLARE
DEFAULT	DEFERRABLE
DEFERRED	DELETE
DESC	DESCRIBE
DESCRIPTOR	DIAGNOSTICS
DISCONNECT	DISTINCT
DOMAIN	DOUBLE
DROP	ELSE
END	END-EXEC
ESCAPE	EXCEPT

EXCEPTION	EXEC
EXECUTE	EXISTS
EXTERNAL	EXTRACT
FALSE	FETCH
FIRST	FLOAT
FOR	FOREIGN
FORTRAN	FOUND
FROM	FULL
GET	GLOBAL
GO	GOTO
GRANT	GROUP
HAVING	HOUR
IDENTITY	IMMEDIATE
IN	INCLUDE
INDEX	INDICATOR
INITIALLY	INNER
INPUT	INSENSITIVE
INSERT	INT
INTEGER	INTERSECT
INTERVAL	INTO
IS	ISOLATION
JOIN	KEY
LANGUAGE	LAST
LEADING	LEFT
LEVEL	LIKE
LOCAl	LOWER
MATCH	MAX
MIN	MINUTE
MODULE	MONTH

NAMES	NATIONAL
NATURAL	NCHAR
NEXT	NO
NONE	NOT
NULL	NULLIF
NUMERIC	OCTET_LENGTH
OF	ON
ONLY	OPEN
OPTION	OR
ORDER	OUTER
OUTPUT	OVERLAPS
PASCAL	POSITION
PRECISION	PREPARE
PRESERVE	PRIMARY
PRIOR	PRIVILEGES
PROCEDURE	PUBLIC
READ	REAL
REFERENCES	RELATIVE
RESTRICT	REVOKE
RIGHT	ROLLBACK
ROWS	SCHEMA
SCROLL	SECOND
SECOND	SECTION
SELECT	SESSION
SESSION_USER	SET
SIZE	SMALLINT
SOME	SPACE
SQL	SQLCA
SQLCODE	SQLERROR

SQLSTATE	SQLWARNING
SUBSTRING	SUM
SYSTEM_USER	TABLE
TEMPORARY	THEN
TIME	TIMESTAMP
TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING
TRANSACTION	TRANSLATE
TRANSLATION	TRIM
TRUE	UNION
UNIQUE	UNKNOWN
UPDATE	UPPER
USAGE	USER
USING	VALUE
VALUES	VARCHAR
VARYING	VIEW
WHEN	WHENEVER
WHERE	WITH
WORK	WRITE
YEAR	ZONE

D

Data Types

ODBC defines the following sets of data types:

- SQL data types, which indicate the data type of data stored at the data source.
- C data types, which indicate the data type of data stored in application buffers.

Each SQL data type corresponds to an ODBC C data type. Before returning data from the data source, the driver converts it to the specified C data type. Before sending data to the data source, the driver converts it from the specified C data type.

This appendix contains the following topics:

- ODBC SQL data types
- ODBC C data types
- Numeric literals
- Data type identifiers including pseudo-type identifiers and Descriptors
- Decimal digits and transfer octet length of SQL data types
- Converting data from SQL to C data types
- Converting data from C to SQL data types

For information about driver-specific SQL data types, see the driver's documentation.

SQL Data Types

In accordance with the SQL-92 standard, each DBMS defines its own set of SQL data types. For each SQL data type in the SQL-92 standard, a #define value, known as a type identifier, is passed as an argument in ODBC functions or returned in the metadata of a result set. Drivers map data source-specific SQL data types to ODBC SQL data type identifiers and driver-specific SQL data type identifiers. The SQL_DESC_CONCISE_TYPE field of an implementation descriptor is where the SQL data type is stored.

ODBC does not support the following SQL_92 data types:

- BIT (ODBC SQL_BIT type has different characteristics)
- BIT_VARYING
- TIME_WITH_TIMEZONE
- TIMESTAMP_WITH_TIMEZONE
- NATIONAL_CHARACTER

C Data Types

ODBC defines the C data types and their corresponding ODBC type identifiers. Applications either call

- **SQLBindCol** or **SQLGetData** to pass an applicable C type identifier in the *TargetType* argument. In this way, applications specify the C data type of the buffer that receives result set data.
- **SQLBindParameter** to pass the appropriate C type identifier in the *ValueType* argument. In this way, application specify the C data type of the buffer containing a statement parameter.

The SQL_DESC_CONCISE_TYPE field of an application descriptor is where the C data type is stored.

Note

Driver-specific C data types do not exist.

Data Type Identifiers

Data type identifiers are stored in the SQL_DESC_CONCISE_TYPE field of a descriptor. Data type identifiers in applications describe their buffers to the driver. They also retrieve metadata about the result set from the driver so applications know what type of C buffers to use for data storage. Applications use data type identifiers to perform these tasks by calling these functions:

- To describe the C data type of application buffers, applications call **SQLBindParameter**, **SQLBindCol**, and **SQLGetData**.
- To describe the SQL data type of dynamic parameters, applications call **SQLBindParameter**.

- To retrieve the SQL data types of result set columns, applications call **SQLColAttribute** and **SQLDescribeCol**.
- To retrieve the SQL data types of parameters, applications call **SQLDescribeParameter**.
- To retrieve the SQL data types of various schema information, applications call **SQLColumns**, **SQLProcedureColumns**, and **SQLSpecialColumns**.
- To retrieve a list of supported data types, applications call **SQLGetTypeInfo**.

In addition, the **SQLSetDescField** and **SQLSetDesRec** descriptor functions are also used to perform the above tasks. For details, see the **SQLSetDescField** and **SQLSetDesRec** functions.

SQL Data Types

A given driver and data source do not necessarily support all of the SQL data types defined in the ODBC grammar. Furthermore, they may support additional, driver-specific SQL data types. A driver's support is determined by the level of SQL-92 conformance. To determine which data types a driver supports, an application calls **SQLGetTypeInfo**. See the *"SQLGetTypeInfo Result Set Example" on page D-6*. For information about driver-specific SQL data types, see the driver's documentation.

A driver also returns the SQL data types when it describes the data types of columns and parameters using the following functions:

- **SQLColAttribute**
- **SQLColumns**
- **SQLDescribeCol**
- **SQLDescribeParam**
- **SQLProcedureColumns**
- **SQLSpecialColumns**

Note

For details on fields that store SQL data type values and characteristics, see *"Data Type Identifiers and Descriptors"* on page D-16.

The following table is not a comprehensive list of SQL data types, but offers commonly used names, ranges, and limits. A data source may only support some of the data types that are listed in the table and depending on your driver, the characteristics of the data types can differ form this table's description. See your driver's documentation for details. The table includes the description of the associated data type from SQL-92 (if applicable)

SQL type identifier [1]	Typical SQL Data Type [2]	Typical Type Description
SQL_CHAR	CHAR(*n*)	Character string of fixed string length *n*.
SQL_VARCHAR	VARCHAR(*n*)	Variable-length character string with a maximum string length *n*.
SQL_LONGVARCHAR	LONG VARCHAR	Variable length character data. Maximum length is data source–dependent. [3]
SQL_WCHAR	WCHAR(*n*)	Unicode character string of fixed string length *n*.
SQL_WVARCHAR	VARWCHAR(*n*)	Unicode variable-length character string with a maximum string length *n*.
SQL_WLONGVARCHAR	LONGWVARCHAR	Unicode variable-length character data. Maximum length is data source-dependent.
SQL_DECIMAL	DECIMAL(*p*,*s*)	Signed, exact, numeric value with a precision *p* and scale *s*. (The maximum precision is driver-defined.) (1 <= *p* <= 15; s <= *p*). [4]
SQL_NUMERIC	NUMERIC(*p*,*s*)	Signed, exact, numeric value with a precision *p* and scale *s*. (1 <= *p* <= 15; s <= *p*). [4]
SQL_SMALLINT	SMALLINT	Exact numeric value with precision 5 and scale 0 (signed: –32,768 <=*n* <= 32,767, unsigned: 0 <= *n* <= 65,535) [5]

SQL_INTEGER	INTEGER	Exact numeric value with precision 10 and scale 0. (signed: -2[31] <= *n* <= 2[31] -1, unsigned: 0 <= *n* <= 2[32] -1) [5]
SQL_REAL	REAL	Signed, approximate, numeric value with a binary precision 24 (zero or absolute value 10[–38] to 10[38]).
SQL_FLOAT	FLOAT(*p*)	Signed, approximate, numeric value with a binary precision of at least *p*. (The maximum precision is driver defined.) [6]
SQL_DOUBLE	DOUBLE PRECISION	Signed, approximate, numeric value with a binary precision 53 (zero or absolute value 10[–308] to 10[308]).
SQL_BIT	BIT	Single bit binary data. [7]
SQL_TINYINT	TINYINT	Exact numeric value with precision 3 and scale 0 (signed: –128 <= *n* <= 127 (unsigned: 0 <= *n* <= 255) [5].
SQL_BIGINT	BIGINT	Exact numeric value with precision 19 (if signed) or 20 (if unsigned) and scale 0 (signed: –2[63] <= *n* <= 2[63] – 1, unsigned: 0 <= *n* <= 2[64] – 1) [3], [5].
SQL_BINARY	BINARY(*n*)	Binary data of fixed length *n*. [3]
SQL_VARBINARY	VARBINARY(*n*)	Variable length binary data of maximum length *n*. The maximum is set by the user. [3]
SQL_LONGVARBINARY	LONG VARBINARY	Variable length binary data. Maximum length is data source–dependent. [3]

SQL_TYPE_DATE [8]	DATE	Year, month, and day fields, conforming to the rules of the Gregorian calendar. (See Constraints of the Gregorian Calendar, later in this appendix.
SQL_TYPE_TIME [8]	TIME(*p*)	Hour, minute, and second fields, with valid values for hours of 00 to 23, valid values for minutes of 00 to 59, and valid values for seconds of 00 to 61. Precision *p* indicates the seconds precision.
SQL_TYPE_TIMESTAMP [8]	TIMESTAMP(*p*)	Year, month, day, hour, minute, and send fields, with valid values as defined for the DATE and Time data types.

Notes

[1] This is the value returned in the DATA_TYPE column by a call to **SQLGetTypeInfo**.

[2] This is the value returned in the NAME and CREATE PARAMS column by a call to **SQLGetTypeInfo**. The NAME column returns the designation-for example, CHAR-while the CREATE PARAMS column returns a comma-separated list of creation parameters such as precision, scale, and length.

[3] This data type has no corresponding data type in SQL-92.

[4] SQL_DECIMAL and SQL_NUMERIC data types differ only in their precision. The precision of a DECIMAL(p,s) is an implementation-defined decimal precision that is no less than p, while the precision of a NUMERIC(p,s) is exactly equal to p.

[5] An application uses **SQLGetTypeInfo** or **SQLColAttribute** to determine if a particular data type or a particular column in a result set is unsigned.

[6] Depending on the implementation, the precision of SQL_FLOAT can be either 24 or 53: if it is 24, the SQL_FLOAT data type is the same as SQL_REAL; if it is 53, the SQL_FLOAT data type is the same as SQL_DOUBLE.

[7] The SQL_BIT data type has different characteristics than the BIT type in SQL-92.

[8] This data type has no corresponding data type in SQL-92.

SQLGetTypeInfo Result Set Example

Applications call **SQLGetTypeInfo** result set for a list of supported data types and their characteristics for a given data source. The example below shows the data types that

SQLGetTypeInfo returns for a data source; all data types under "DATA_TYPE" are supported in this data source.

TYPE_NAME	DATA_TYPE	COLUMN_SIZE	LITERAL_ PREFIX	LITERAL_ SUFFIX	CREATE_ PARAMS	NULLABLE
"char"	SQL_CHAR	255	""	""	"length"	SQL_TRUE
"text"	SQL_LONG VARCHAR	2147483647	""	""	Null	SQL_TRUE
"decimal"	SQL_ DECIMAL	28	<Null>	<Null>	"precision, scale"	SQL_TRUE
"real"	SQL_REAL	7	<Null>	<Null>	<Null>	SQL_TRUE
"datetime"	SQL_TYPE_ TIMESTAMP	23	""	""	<Null>	SQL_TRUE

	CASE_SENSI TIVE	SEARCHABLE	UNSIGNED_ ATTRIBUTE	FIXED_ PREC_ SCALE	AUTO_ UNIQUE_ VALUE	LOCAL_ TYPE_ NAME
SQL_CHAR	SQL_FALSE	SQL_SEARCH-ABLE	<Null>	SQL_FALSE	<Null>	"char"
SQL_LONG VARCHAR	SQL_FALSE	SQL_PRED_CHAR	<Null>	SQL_FALSE	<Null>	"text"
SQL_ DECIMAL	SQL_FALSE	SQL_PRED_BASIC	SQL_FALSE	SQL_FALSE	SQL_FALSE	"decimal"
SQL_REAL	SQL_FALSE	SQL_PRED_BASIC	SQL_FALSE	SQL_FALSE	SQL_FALSE	"real"
SQL_TYPE_ TIMESTAMP	SQL_FALSE	SQL_SEARCH-ABLE	<Null>	SQL_FALSE	<Null>	"datetime"

	MINIMUM_ SCALE	MAXIMUM_ SCALE	SQL_DATA_ TYPE	SQL_DATE TIME_SUB	NUM_ PREC_ RADIX	INTERVAL_ PRECISION
SQL_CHAR	<Null>	<Null>	SQL_CHAR	<Null>	<Null>	<Null>
SQL_LONG VARCHAR	<Null>	<Null>	SQL_LONG VARCHAR	<Null>	<Null>	<Null>
SQL_ DECIMAL	0	28	SQL_ DECIMAL	<Null>	10	<Null>
SQL_REAL	<Null>	<Null>	SQL_REAL	<Null>	10	<Null>
SQL_TYPE_ TIMESTAMP	3	3	SQL_DATETIM E	SQL_CODE _TIMESTA MP	<Null>	12

C Data Types

The ODBC Driver supports all C data types in keeping with the need for character SQL type conversion to and from all C types.

The C data type is specified in the following functions:

- **SQLBindCol** and **SQLGetData** functions with the *TargetType* argument.
- **SQLBindParameter** with the *ValueType* argument.
- **SQLSetDescField** to set the SQL_DESC_CONCISE_TYPE field of an ARD or APD
- **SQLSetDescRec** with the *Type* argument, *SubType* argument (if needed), and the *DescriptorHandle* argument set to the handle of an ARD or APD.

The table below contains C type identifiers for the C data types, as well as the ODBC C data type that is associated with each identifier and C type definition.

C Type Identifier	ODBC C Typedef	C Type
SQL_C_CHAR	SQLCHAR *	unsigned char
SQL_C_SSHORT [h]	SQLSMALLINT	short int
SQL_C_USHORT [h]	SQLUSMALLINT	unsigned short int
SQL_C_SLONG [h]	SQLINTEGER	long int
SQL_C_ULONG [h]	SQLUINTEGER	unsigned long int

SQL_C_FLOAT	SQLREAL	float
SQL_C_DOUBLE	SQLDOUBLE SQLFLOAT	double
SQL_C_STINYINT	SCHAR	signed char
SQL_C_UTINYINT	UCHAR	unsigned char
SQL_C_SBIGINT	SQLBIGINT	_int64 [g]
SQL_C_UBIGINT	SQLUBIGINT	unisigned _int64 [g]
SQL_C_BINARY	SQLCHAR *	unsigned char *
SQL_C_TYPE_DATE [c]	SQL_DATE_STRUCT	struct tagDATE_STRUCT{ SQLSMALLINT year; SQLSMALLING month; SQLUSMALLINT day; } DATE_STRUCT; [a]

SQL_C_TIME	TIME_STRUCT	struct tagTIME_STRUCT {
		SQLUSMALLINT hour;
		SQLUSMALLINT minute;[d]
		SQLUSMALLINT second;[e]
		}

SQL_C_TIMESTAMP	TIMESTAMP_STRUCT	struct tagTIMESTAMP_STRUCT
		{
		SQLSMALLINT year; [a]
		SQLUSMALLINT month; [b]
		SQLUSMALLINT day; [c]
		SQLUSMALLINT hour;
		SQLUSMALLINT minute; [d]
		SQLUSMALLINT second;[e]
		SQLUINTEGER fraction; [f]
		}

SQL_C_STINYINT	SCHAR	signed char
SQL_C_UTINYINT	UCHAR	unsigned char
SQL_C_BINARY	UCHAR FAR *	unsigned char FAR *

SQL_C_DATE	DATE_STRUCT	struct tagDATE_STRUCT {
		SQLSMALLINT year; [a]
		SQLUSMALLINT month; [b]
		SQLUSMALLINT day; [c]
		}

SQL_C_TIME	TIME_STRUCT	struct tagTIME_STRUCT {
		SQLUSMALLINT hour;
		SQLUSMALLINT minute; [d]
		SQLUSMALLINT second; [e]
		}

Notes

[a] The values of the year, month, day, hour, minute, and second fields in the datetime C data types must conform to the constraints of the Gregorian calendar. (See *"Constraints of the Gregorian Calendar" on page D-21*.)

[b] The value of the fraction field is the number of billionths of a second and ranges from 0 through 999,999,999 (1 less than 1 billion). For example, the value of the fraction field for a half-second is 500,000,000, for a thousandth of a second (one millisecond) is 1,000,000, for a millionth of a second (one microsecond) is 1,000, and for a billionth of a second (one nanosecond) is 1.

[c] In ODBC 2.x, the C date, time, and timestamp data types are SQL_C_DATE, SQL_C_TIME, and SQL_C_TIMESTAMP.

[d] A number is stored in the val field of the SQL_NUMERIC_STRUCT structure as a scaled integer, in little endian mode (the leftmost byte being the least-significant byte). For example, the number 10.001 base 10, with a scale of 4, is scaled to an integer of 100010. Because this is 186AA in hexadecimal format, the value in SQL_NUMERIC_STRUCT would be "AA 86 01 00 00 … 00", with the number of bytes defined by the SQL_MAX_NUMERIC_LEN #define.

[e] The precision and scale fields of the SQL_C_NUMERIC data type are never used for input from an application, only for output from the driver to the application. When the driver writes a numeric value into the SQL_NUMERIC_STRUCT, it will use its own driver-specific default as the value for the precision field, and it will use the value in the SQL_DESC_SCALE field of the application descriptor (which defaults to 0) for the scale field. An application can provide its own values for precision and scale by setting the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the application descriptor.

[f] The sign field is 1 if positive, 0 if negative.

[g] _int64 might not be supplied by some compilers.

[h] _SQL_C_SHORT, SQL_C_LONG, and SQL_C_TINYINT have been replaced in ODBC by signed and unsigned types: SQL_C_SSHORT and SQL_C_USHORT, SQL_C_SLONG and SQL_C_ULONG, and SQL_C_STINYINT and SQL_C_UTINYINT. An ODBC 3.x driver that should work with ODBC 2.x applications should support

SQL_C_SHORT, SQL_C_LONG, and SQL_C_TINYINT, because when they are called, the Driver Manager passes them through to the driver.

64-Bit Integer Structures

The C data type identifiers SQL_C_SBIGINT and SQL_C_UBIGINT used on Microsoft C compilers is _int64. When a non-Microsoft C compiler is used, the C type may differ. If the compiler in use is supporting 64-bit integers natively, then define the driver or application ODBCINT64 as the native 64-bit integer type. If compiler in use does not support 64-bit integers natively, define the following structures to ensure access to these C types:

```
typedef struct{
SQLUINTEGER dwLowWord;
SQLUINTEGER dwHighWord;
} SQLUBIGINT
typedef struct {
SQLUINTEGER dwLowWord;
SQLINTEGER sdwHighWord;
} SQLBIGINT
```

Because a 64-bit integer is aligned to the 8-byte boundary, be sure to align these structures to an 8-byte boundary.

Default C Data Types

In applications that specify SQL_C_DEFAULT in **SQLBindCol**, **SQLGetData**, or **SQLBindParameter**, the driver assumes that the C data type of the output or input buffer corresponds to the SQL data type of the column or parameter to which the buffer is bound.

! Important

If the application is interoperable, do not use the SQL_C_DEFAULT. Instead, specify the C type of the buffer in use.

Drivers cannot always determine the correct default C type for these reasons:

- The DBMS may have promoted a SQL data type of a column or a parameter; in this case, the driver is unable to determine the original SQL data type and consequently, cannot determine the corresponding default C data type.

- The DBMS determined whether the data type of a column or parameter is signed or unsigned; in this case, the driver is unable to determine this for a particular SQL data type and consequently, cannot determine this for the corresponding default C data type.

See*"Converting Data from SQL to C Data Types"* on page D-21.

SQL_C_TCHAR

The SQL_C_TCHAR type identifier is used for unicode purposes. Use this identifier in applications that transfer character data and are compiled as both ANSI and Unicode. Note that the SQL_C_TCHAR is not a type identifier in the conventional sense; instead, it is a macro contained in the header file for Unicode conversion. SQL_C_CHAR or SQL_C_WCHAR replaces SQL_C_TCHAR depending on the setting of the UNICODE #define.

Numeric Literals

To store numeric data values in character strings, you use numeric literals. Numeric literal syntax specifies what is stored in the target during the following conversions:

- SQL data to a SQL_C_CHAR string
- C data to a SQL_CHAR or SQL_VARCHAR string

The syntax also validates what is stored in the source during the following conversions:

- numeric stored as a SQL_C_CHAR string to numeric SQL data
- numeric stored as a SQL_CHAR string to numeric C data

See the numeric literal syntax described in *Appendix C, "SQL Minimum Grammar"* for details.

Conversion Rules

The rules in this section apply to conversions involving numeric literals. Following are terms used in this section:

Term	Meaning
Store assignment	Refers to sending data into a table column in a database when calling **SQLExecute** and **SQLExecDirect**. During store assignment, "target" refers to a database column and "source" refers to data in application buffers.

Term	Meaning
Retrieval assignment	Refers to retrieving data from the database into application buffers when calling **SQLFetch**, **SQLGetData**, and **SQLFetchScroll**. During retrieval assignment, "target" refers to the application buffers and "source" refers to the database column.
CS	Value in the character source.
NT	Value in the numeric target.
NS	Value in the numeric source.
CT	Value in the character target.
Precision of an exact numeric literal	Number of digits that the literal contains.
Scale of an exact numeric literal	Number of digits to the right of the expressed or implied period.
Precision of an approximate numeric literal	Precision of the literal's mantissa.

Rules for Character Source to Numeric Target

Following are the rules for converting from a character source (CS) to a numeric target (NT):

1. Replace CS with the value obtained by removing any leading or trailing spaces in CS. If CS is not a valid numeric-literal, SQLSTATE 22018 (Invalid character value for cast specification) is returned.
2. Replace CS with the value obtained by removing leading zeroes before the decimal point, trailing zeroes after the decimal point, or both.
3. Convert CS to NT. If the conversion results in a loss of significant digits, SQLSTATE 22003 (Numeric value out of range) is returned. If the conversion results in the loss of nonsignificant digits, SQLSTATE 01S07 (Fractional truncation) is returned.

Rules for Numeric Source to Character Target

Following are the rules for converting from a numeric source (NS) to a character target (CT):

1. Let LT be the length in characters of CT.

   For retrieval assignment, LT is equal to the length of the buffer in characters minus the number of bytes in the null-termination character for this character set.

2. Take one the following actions depending on the type of NS.

- If NS is an exact numeric type, then let YP equal the shortest character string that conforms to the definition of exact-numeric-literal such that the scale of YP is the same as the scale of NS, and the interpreted value of YP is the absolute value of NS.
- If NS is an approximate numeric type, then let YP be a character string as follows:

Case:

a. If NS is equal to 0, then YP is 0.

b. Let YSN be the shortest character string that conforms to the definition of exact-numeric-literal and whose interpreted value is the absolute value of NS. If the length of YSN is less than the (precision + 1) of the data type of NS, then let YP equal YSN.

c. Otherwise, YP is the shortest character string that conforms to the definition of approximate-numeric-literal whose interpreted value is the absolute value of NS and whose mantissa consists of a single digit that is not '0', followed by a period and an unsigned-integer.

3. If NS is less than 0, then let Y be the result of:

'-' || YP

where '||' is the string concatenation operator.

Otherwise, let Y equal YP.

4. Let LY be the length in characters of Y.

5. Take one of the following action depending on the value of LY.

- If LY equals LT, then CT is set to Y.
- If LY is less than LT, then CT is set to Y extended on the right by appropriate number of spaces.
- Otherwise (LY > LT), copy the first LT characters of Y into CT.

Case:

- If this is a store assignment, return the error SQLSTATE 22001 (String data, right-truncated).
- If this is retrieval assignment, return the warning SQLSTATE 01004 (String data, right-truncated). When the copy results in the loss of fractional digits (other than trailing zeros), depending on the driver definition, one of the following actions occurs:

a. The driver truncates the string in Y to an appropriate scale (which can be zero also) and writes the result into CT.

b. The driver rounds the string in Y to an appropriate scale (which can be zero also) and writes the result into CT.

c. The driver neither truncates nor rounds, but just copies the first LT characters of Y into CT.

Overriding Default Precision and Scale for Numeric Data Types

The following table provides the override default precision and scale values for numeric data type.

Function calls to	Setting	Override
SQLBindCol or **SQLSetDescField**	SQL_DESC_TYPE field in an ARD is set to SQL_C_NUMERIC	SQL_DESC_SCALE field in the ARD is set to 0 and the SQL_DESC_PRECISION field is set to a driver-defined default precision.[a]
SQLBindParameter or **SQLSetDescField**	SQL_DESC_SCALE field in an APD is set to SQL_C_NUMERIC	SQL_DESC_SCALE field in the ARD is set to 0 and the SQL_DESC_PRECISION field is set to a driver-defined default precision. This is true for input, input/output, or output parameters.[a]
SQLGetData	Data is returned into a SQL_C_NUMERIC structure	Default SQL_DESC_SCALE and SQL_DESC_PRECISION fields are used.[b]

Notes

[a] If the defaults are not acceptable for an application, the application can call the **SQLSetDescField** or **SQLSetDescRec** to set the SQL_DESC_SCALE or SQL_DESC_PRECISION field.

[b] If the defaults are not acceptable, the application must call **SQLSetDescRec** or **SQLSetDescField** to set the fields and then call **SQLGetData** with a *TargetType* of SQL_ARD_TYPE to use the values in the descriptor fields.

Data Type Identifiers and Descriptors

Unlike the "concise" SQL and C data types, where each identifier refers to a single data type, descriptors do not in all cases use a single value to identify data types. In some cases, descriptors use a verbose data type and a type subcode. For most data types, the verbose data type identifier matches the concise type identifier.

The exception, however, is the datetime and interval data types. For these data types:

- SQL_DESC_TYPE contains the verbose type (SQL_DATETIME)
- SQL_DESC_CONCISE_TYPE contains a concise type

For details on setting fields and a settings affect on other fields, see the **SQLSetDescField** function description on the Microsoft ODBC Website.

When the SQL_DESC_TYPE or SQL_DESC_CONCISE_TYPE field is set for some data types, the following fields are set to default values appropriate for the data type:

- SQL_DESC_DATETIME_INTERVAL_PRECISION
- SQL_DESC_LENGTH
- SQL_DESC_PRECISION
- SQL_DESC_SCALE

For more information, see the SQL_DESC_TYPE field under **SQLSetDescField** function description on the Microsoft ODBC Website.

Note

If the default values set are not appropriate, you can explicitly set the descriptor field in the application by calling **SQLSetDescField**.

The following table lists for each SQL and C type identifier, the concise type identifier, verbose identifier, and type subcode for each datetime.

For datetime data types, the SQL_DESC_TYPE have the same manifest constants for both SQL data types (in implementation descriptors) and for C data types (in application descriptors):

Concise SQL Type	Concise C Type	Verbose Type	DATETIME_ INTERVAL_CODE
SQL_TYPE_ DATE	SQL_C_TYPE_ DATE	SQL_DATETIME	SQL_CODE_DATE
SQL_TYPE_TIME	SQL_C_TYPE_ TIME	SQL_DATETIME	SQL_CODE_TIME
SQL_TYPE_ TIMESTAMP	SQL_C_TYPE_ TIMESTAMP	SQL_DATETIME	SQL_CODE_TIME STAMP

Pseudo-Type Identifiers

ODBC defines a number of pseudo-type identifiers, which depending on the situation, resolve to existing data types. Note that these identifiers do not correspond to actual data types, but are provided for your application programming convenience.

Decimal Digits

Decimal digits apply to decimal and numeric data types. They refer to the maximum number of digits to the right of the decimal point, or the scale of the data. Because the number of digits to the right of decimal point is not fixed, the scale is undefined for approximate floating-point number columns or parameters. When datetime data contains a seconds component, the decimal digits are the number of digits to the right of the decimal point in the seconds component of the data.

Typically, the maximum scale matches the maximum precision for SQL_DECIMAL and SQL_NUMERIC data types. Some data sources, however, have their own maximum scale limit. An application can call **SQLGetTypeInfo** to determine the minimum and maximum scales allowed for a data type.

The following ODBC functions return parameter decimal attributes in a SQL statement data type or decimal attributes on a data source:

ODBC Function	Returns...
SQLDescribeCol	Decimal digits of the columns it describes.
SQLDescribeParam	Decimal digits of the parameters it describes.
SQLProcedureColumns	Decimal digits in a column of a procedure.

ODBC Function	Returns...
SQLColumns	Decimal digits in specified tables (such as the base table, view, or a system table).
SQLColAttribute	Decimal digits of columns at the data source.
SQLGetTypeInfo	Minimum and maximum decimal digits of a SQL data type on a data source.

Note that **SQLBindParameter** sets the decimal digits for a parameter in a SQL statement.

The values returned by ODBC functions for decimal digits correspond to "scale" as defined in ODBC 2.x.

Descriptor fields describe the characteristics of a result set. They do not contain valid data values before statement execution. However, the decimal digits values returned by **SQLColumns**, **SQLProcedureColumns**, and **SQLGetTypeInfo**, do represent the characteristics of database objects, such as table columns and data types form the data source's catalog.

Each concise SQL data type has the following decimal digits definition as noted in the table below.

SQL Type Identifier	Decimal Digits
All character and binary types [a]	N/A
SQL_DECIMAL SQL_NUMERIC	The defined number of digits to the right of the decimal point. For example, the scale of a column defined as NUMERIC(10,3) is 3. This can be a negative number to support storage of very large numbers without using exponential notation; for example, "12000" could be stored as "12" with a scale of -3.
All exact numeric types other than SQL_DECIMAL and SQL_NUMERIC [a]	0
All approximate data types [a]	N/A
SQL_TYPE_DATE, and all interval types with no seconds component [a]	The number of digits to the right of the decimal point in the seconds part of the value (fractional seconds). This number cannot be negative.

Notes

[a] **SQLBindParameter**'s *DecimalDigits* argument is ignored for this data type.

For decimal digits, the values returned do not correspond to the values in any one descriptor field. The values returned (for example, in **SQLColAttribute**) for the decimal digits can come from either the SQL_DESC_SCALE or the SQL_DESC_PRECISION field, depending on the data type, as shown in the following table:

SQL Type Identifier	Descriptor field corresponding to decimal digits
All character and binary types	N/A
All exact numeric types	SCALE
All approximate numeric types	N/A
All datetime types	PRECISION

Transfer Octet Length

When data is transferred to its default C data type, an application receives a maximum number of bytes. This maximum is known as the transfer octet length of a column. For character data, space for the null-termination character is not included in the transfer octet length. Note that the transfer octet length in bytes can differ from the number of bytes needed to store the data on the data source.

The following ODBC functions return parameter decimal attributes in a SQL statement data type or decimal attributes on a data source:

ODBC Function	Returns
SQLColumns	Transfer octet length of a column in specified tables (such as the base table, view, or a system table).
SQLColAttribute	Transfer octet length of columns at the data source.
SQLProcedureColumns	Transfer octet length of a column in a procedure.

The values returned by ODBC functions for the transfer octet length may not correspond to the values returned in SQL_DESC_LENGTH. For all character and binary types, the values come from a descriptor field's SQL_DESC_OCTET_LENGTH. For other data types, there is no descriptor field that stores this information.

Descriptor fields describe the characteristics of a result set. They do not contain valid data values before statement execution. In its result set, **SQLColAttribute** returns the transfer octet length of columns at the data source; these values may not match the values in the SQL_DESC_OCTET_LENGTH descriptor fields. For more information on descriptor fields, see **SQLSetDescField** function description on the Microsoft ODBC Website.

Each concise SQL data type has the following transfer octet length definition as noted in the table below.

SQL Type Identifier	Transfer Octet Length
All character and binary types [a]	The defined or the maximum (for variable type) length of the column in bytes. This value matches the one in the SQL_DESC_OCTET_LENGTH descriptor field.
SQL_DECIMAL SQL_NUMERIC	The number of bytes required to hold the character representation of this data if the character set is ANSI, and twice this number if the character set is UNICODE. The character representation is the maximum number of digits plus two; the data is returned as a character string, where the characters are needed for digits, a sign, and a decimal point. For example, the transfer length of a column defined as NUMERIC(10,3) is 12.
SQL_TINYINT	1
SQL_SMALLINT	2
SQL_INTEGER	4
SQL_BIGINT	The number of bytes required to hold the character representation of this data if the character set is ANSI, and twice this number if the character set is UNICODE. This data type is returned as a character string by default. The character representation consists of 20 characters for 19 digits and a sign (if signed), or 20 digits (if unsigned). and a decimal point. The length is 20.
SQL_REAL	4
SQL_FLOAT	8
SQL_DOUBLE	8
All binary types [a]	The number of bytes required to store the defined (for fixed types) or maximum (for variable types) number of characters.
SQL_TYPE_DATE SQL_TYPE_TIME	6 (size of the structures SQL_DATE_STRUCT or SQL_TIME_STRUCT).
SQL_TYPE_TIMESTAMP	16 (size of the structure SQL_TIMESTAMP_STRUCT).

Notes

[a] SQL_NO_TOTAL is returned when the driver cannot determine the column or parameter length for variable types.

Constraints of the Gregorian Calendar

The following table are the Gregorian calendar constraints for date and datetime data types.

Value	Requirement
month field	Must be between 1 and 12, inclusive.
day field	Range must be from 1 through the number of days in the month, which is determined from the values of the year and months fields and can be 28, 29, 30, or 31. A leap year can also affect the number of days in the month.
hour field	Must be between 0 and 23, inclusive.
minute field	Must be between 0 and 59, inclusive.
trailing seconds field	Must be between 0 and 61.9(*n*), inclusive, where *n* specifies the number of "9" digits and the value of *n* is the fractional seconds precision. (The range of seconds permits a maximum of two leap seconds to maintain synchronization of sidereal time.)

Converting Data from SQL to C Data Types

When an application calls **SQLFetch**, **SQLFetchScroll**, or **SQLGetData**, the driver retrieves the data from the data source. If necessary, it converts the data from the data type in which the driver retrieved it to the data type specified by the *TargeType* argument in **SQLBindCol** or **SQLGetData**. Finally, it stores the data in the location pointed to by the *TargetValuePtr* argument in **SQLBindCol** or **SQLGetData** (and the SQL_DESC_DATA_PTR field of the ARD).

The following table shows the supported conversions from ODBC SQL data types to ODBC C data types. A solid circle indicates the default conversion for a SQL data type (the C data type to which the data will be converted when the value of *TargetType* is SQL_C_DEFAULT). A hollow circle indicates a supported conversion.

For an ODBC 3.x application working with an ODBC 2.x driver, conversion from driver-specific data types might not be supported.

The format of the converted data is not affected by the Microsoft Windows country setting.

C Data Type—SQL_C_*datatype* where *datatype* is:

SQL Data Type	CHAR	WCHAR	NUMERIC	STINYINT	UTINYINT	TINYINT	SBIGINT	UBIGINT	SSHORT	USHORT	SHORT	SLONG	ULONG	LONG	FLOAT	DOUBLE	BINARY	*DATE	*TIME	*TIMESTAMP
SQL_CHAR	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_VARCHAR	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_LONGVAR CHAR	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WCHAR	o	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WVARCHAR	o	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WLONG VARCHAR	o	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_DECIMAL	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_NUMERIC	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_TINYINT (signed)	o	o	o	•	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_TINYINT (unsigned)	o	o	o	o	•	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_SMALLINT (signed)	o	o	o	o	o	o	o	o	•	o	o	o	o	o	o	o	o			
SQL_SMALLINT (unsigned)	o	o	o	o	o	o	o	o	o	•	o	o	o	o	o	o	o			
SQL_INTEGER(signed)	o	o	o	o	o	o	o	o	o	o	o	•	o	o	o	o	o			
SQL_INTEGER (unsigned)	o	o	o	o	o	o	o	o	o	o	o	o	•	o	o	o	o			
SQL_BIGINT (signed)	o	o	o	o	o	o	•	o	o	o	o	o	o	o	o	o	o			
SQL_BIGINT (unsigned)	o	o	o	o	o	o	o	•	o	o	o	o	o	o	o	o	o			
SQL_REAL	o	o	o	o	o	o	o	o	o	o	o	o	o	o	•	o	o			
SQL_FLOAT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	•	o			
SQL_DOUBLE	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	•	o			
SQL_BINARY	o	o															•			
SQL_VARBINARY	o	o															•			
SQL_LONG VARBINARY	o	o															•			
SQL_TYPE_DATE	o	o															o	•		o
SQL_TYPE_TIME	o	o															o		•	o
SQL_TYPE_TIMEST AMP	o	o															o	o	o	•

• Default conversion o Supported conversion

* These datatypes have "TYPE" in the datatype name. For example, SQL_C_TYPE_DATE, SQL_C_TYPE_TIME, and SQL_C_TYPE_TIMESTAMP

Table Description—SQL to C

The tables in the following sections describe how the driver or data source converts data retrieved from the data source; drivers are required to support conversions to all ODBC C data types from the ODBC SQL data types that they support. For a given ODBC SQL data type, the first column of the table lists the legal input values of the *TargetType* argument in **SQLBindCol** and **SQLGetData**. The second column lists the outcomes of a test, often using the *BufferLength* argument specified in **SQLBindCol** or **SQLGetData**, which the driver performs to determine if it can convert the data. For each outcome, the third and fourth columns list the values placed in the buffers specified by the *TargetValuePtr* and *StrLen_or_IndPtr* arguments specified in **SQLBindCol** or **SQLGetData** after the driver has attempted to convert the data. (The *StrLen_or_IndPtr* argument corresponds to the SQL_DESC_OCTET_LENGTH_PTR field of the ARD.) The last column lists the SQL-STATE returned for each outcome by **SQFetch**, **SQLFetchScroll**, or **SQLGetData**.

If the *TargetType* argument in **SQLBindCol** or **SQLGetData** contains a value for an ODBC C data type not shown in the table for a given ODBC SQL data type, **SQLFetch**, **SQLFetchScroll**, or **SQLGetData** returns SQLSTATE 07006 (Restricted data type attribute violation). If the *TargetType* argument contains a value that specifies a conversion from a driver-specific SQL data type to an ODBC C data type and this conversion is not supported by the driver, **SQLFetch**, **SQLFetchScroll**, or **SQLGetData** returns SQLSTATE HYC00 (Optional feature not implemented).

Although it is not shown in the tables, the driver returns SQL_NULL_DATA in the buffer specified by the *StrLen_or_IndPtr* argument when the SQL data value is NULL. For an explanation of the use of *StrLen_or_IndPtr* does not include the null-termination byte. If *TargetValuePtr* is a null pointer, **SQLGetData** returns SQLSTATE HY009 (Invalid use of null pointer); in **SQLBindCol**, this unbinds the columns.

The following terms and conventions are used in the tables:

- **Byte length of data** is the number of bytes of C data available to return in **TargetValuePtr*, whether or not the data will be truncated before it is returned to the application. For string data, this does not include the space for the null-termination character.
- **Character byte length** is the total number of bytes needed to display the data in character format.
- Words in *italics* represent function arguments or elements of the SQL grammar. See *Appendix C, "SQL Minimum Grammar"* for the syntax of grammar elements,

SQL to C: Character

The character ODBC SQL data types are:

SQL_CHAR
SQL_VARCHAR
SQL_LONGVARCHAR
SQL_WCHAR
SQL_WVARCHAR
SQL_WLONGVARCHAR

The following table shows the ODBC C data types to which character SQL data may be converted. For an explanation of the columns and terms in the table, see the “Table Description—SQL to C” on page D-23.

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQL-STATE
SQL_C_CHAR	Byte length of data < *BufferLength*	Data	Length of data in bytes	N/A
	Byte length of data >= *BufferLength*	Truncated data	Length of data in bytes	01004
SQL_C_WCHAR	Character length of data < *BufferLength*	Data	Length of data in characters	N/A
	(Character length of data) >= *BufferLength*	Truncated data	Length of data in characters	01004
SQL_C_STINYINT SQL_C_UTINYINT SQL_C_TINYINT SQL_C_SBIGINT SQL_C_UBIGINT SQL_C_SSHORT SQL_C_USHORT SQL_C_SHORT SQL_C_SLONG SQL_C_ULONG SQL_C_LONG SQL_C_NUMERIC	Data converted without truncation [b]	Data	Number of bytes of the C data type	N/A
	Data converted with truncation of fractional digits[a]	Truncated data	Number of bytes of the C data type	01S07
	Conversion of data would result in loss of whole (as opposed to fractional) digits [b]	Undefined	Undefined	22003
	Data is not a *numeric-literal* [b]	Undefined	Undefined	22018

SQL_C_FLOAT SQL_C_DOUBLE	Data is within the range of the data type to which the number is being converted [a]	Data	Size of the C data type	N/A
		Undefined		22003
	Data is outside the range of the data type to which the number is being converted [a]	Undefined	Undefined	22018
	Data is not a *numeric-literal* [b]			
SQL_C_BINARY	Byte length of data <= *BufferLength*	Data	Length of data	N/A
		Truncated data		01004
	Byte length of data > *BufferLength*		Length of data	
SQL_C_TYPE_DATE	Data value is a valid *date-value* [a]	Data	6 [b]	N/A
		Data	6 [b]	N/A
	Data value is a valid *timestamp-value;* time portion is zero [a]	Truncated data	6 [b]	01S07
		Undefined	Undefined	22018
	Data value is a valid *timestamp-value*; time portion is nonzero [a], [c],			
	Data value is not a valid *date-value or timestamp_value* [a]			

SQL_C_TYPE_TIME	Data value is a valid *time-value and the fractional seconds value is 0* [a]	Data	6 [b]	N/A
		Data	6 [b]	N/A
	Data value is a valid *timestamp-value* or a *valid time_value;* fractional seconds portion is zero portion is zero[a], [d]	Truncated data	6 [b]	01S07
		Undefined	Undefined	22018
	Data value is a valid *timestamp-value*; fractional seconds portion is nonzero [a], [d], [e]			
	Data value is not a valid *timestamp-value* or *time_value* [a]			
SQL_C_TYPE TIMESTAMP	Data value is a valid *timestamp-value* or a *valid time_value;* fractional seconds portion not truncated [a], [d]	Data	16 [b]	N/A
		Truncated data	16 [b]	01S07
	Data value is a valid *timestamp-value or a valid time_value;* fractional seconds portion truncated [a]	Data [f]	16 [b]	N/A
		Data [g]	16 [b]	N/A
		Undefined	Undefined	22018
	Data value is a valid *date-value*[a]			
	Data value is a valid *time_value* [a]			
	Data value is not a valid *date_value*, *time_value*, or *timestamp_value* [a]			

Notes

[a] The value of *BufferLength* is ignored for this conversion. The driver assumes that the size of **TargetValuePtr* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The time portion of the *timestamp-value* is truncated.

[d] The date portion of the *timestamp-value* is ignored.

[e] The fractional seconds portion of the timestamp is truncated.

[f] The time fields of the timestamp structure are set to zero.

[g] The date fields of the timestamp structure are set to the current date.

When character SQL data is converted to numeric, date, time, or timestamp C data, leading and trailing spaces are ignored.

SQL to C: Numeric

The numeric ODBC SQL data types are:

SQL_DECIMAL SQL_BIGINT
SQL_NUMERIC SQL_REAL
SQL_TINYINT SQL_FLOAT
SQL_SMALLINT SQL_DOUBLE
SQL_INTEGER

The following table shows the ODBC C data types to which numeric SQL data may be converted. For an explanation of the columns and terms in the table, see page the "Table Description—SQL to C" on page D-23.

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_Ind Prt	SQL-STATE
SQL_C_CHAR	Character byte length < *BufferLength*	Data	Length of data in bytes	N/A
		Truncated data		01004
		Undefined	Length of data in bytes	22003
	Number of whole (as opposed to fractional) digits < *BufferLength*		Undefined	
	Number of whole (as opposed to fractional) digits ≥ *BufferLength*			
SQL_C_WCHAR	Character length < *BufferLength*	Data	Length of data in characters	N/A
		Truncated data		01004
		Undefined	Length of data in characters	22003
	Number of whole (as opposed to fractional) digits < *BufferLength*		Undefined	
	Number of whole (as opposed to fractional) digits ≥ *BufferLength*			

SQL_C_STINYINT SQL_C_UTINYINT SQL_C_TINYINT SQL_C_SBIGINT SQL_C_UBIGINT SQL_C_SSHORT SQL_C_USHORT SQL_C_SHORT a SQL_C_SLONG SQL_C_ULONG SQL_C_LONG SQL_C_NUMERIC	Data converted without truncation [a]	Data	Size of the C data type	N/A
	Data converted with truncation of fractional digits [a]	Truncated data	Size of the C data type	01S07
	Conversion of data would result in loss of whole (as opposed to fractional) digits [a]	Undefined		22003
SQL_C_FLOAT SQL_C_DOUBLE	Data is within the range of the data type to which the number is being converted [a]	Data	Size of the C data type	N/A
	Data is outside the range of the data type to which the number is being converted [a]	Undefined	Undefined	22003
SQL_C_BINARY	Length of data ≤ *BufferLength*	Data	Length of data	N/A |
	Length of data > *BufferLength*	Undefined	Undefined	22003

Notes

[a]The value of *BufferLength* is ignored for this conversion. The driver assumes that the size of **TargetValuePtr* is the size of the C data type.

[b] This is the size of the corresponding C data type.

SQL to C: Binary

The binary ODBC SQL data types are:

SQL_BINARY
SQL_VARBINARY
SQL_LONGVARBINARY

The following table shows the ODBC C data types to which binary SQL data may be converted. For an explanation of the columns and terms in the table, see the "Table Description—SQL to C" on page D-23.

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQL-STATE
SQL_C_CHAR	(Byte length of data) * 2 < *BufferLength*	Data	Length of data in bytes	N/A
	(Byte length of data) * 2 >= *BufferLength*	Truncated data	Length of data in bytes	01004
SQL_C_WCHAR	(Character length of data) * 2 < *Buffer-Length*	Data	Length of data in characters	N/A
	(Character length of data) * 2 >= *Buffer-Length*	Truncated data	Length of data in characters	01004
SQL_C_BINARY	Byte length of data <= *BufferLength*	Data	Length of data in bytes	N/A
	Byte Length of data > *BufferLength*	Truncated data	Length of data in bytes	01004

When binary SQL data is converted to character C data, each byte (8 bits) of source data is represented as two ASCII characters. These characters are the ASCII character representation of the number in its hexadecimal form. For example, a binary 00000001 is converted to "01" and a binary 11111111 is converted to "FF".

The driver always converts individual bytes to pairs of hexadecimal digits and terminates the character string with a null byte. Because of this, if *BufferLength* is even and is less than the length of the converted data, the last byte of the **TargetValuePtr* buffer is not used. (The converted data requires an even number of bytes, the next-to-last byte is a null byte, and the last byte cannot be used.)

Application developers are discouraged from binding binary SQL data to a character C data type. This conversion is usually inefficient and slow.

SQL to C: Date

The date ODBC SQL data type is:

SQL_DATE

The following table shows the ODBC C data types to which date SQL data may be converted. For an explanation of the columns and terms in the table, see the "Table Description—SQL to C" on page D-23.

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	*BufferLength* > Character byte length	Data	10	N/A
	11<= *BufferLength* <= Character byte length	Truncated data	Length of data in bytes	01004
	BufferLength < 11	Undefined	Undefined	22003
SQL_C_WCHAR	*BufferLength* > Character length	Data	10	N/A
	11<= *BufferLength* <= Character length	Truncated data	Length of data in bytes	01004
	BufferLength < 11	Undefined	Undefined	22003

SQL_C_BINARY	Byte length of data <= *BufferLength* >	Data	Length of data in bytes	N/A
	Character byte length	Undefined		22003
			Undefined	
	Byte length of data <= B*ufferLength*			
SQL_C_DATE	None [a]	Data	6 [c]	N/A
SQL_C_TIMESTAMP	None [a]	Data [b]	16 [c]	N/A

Notes

[a] The value of *BufferLength* is ignored for this conversion. The driver assumes that the size of **TargetValuePtr* is the size of the C data type.

[b] The time fields of the timestamp structure are set to zero.

[c] This is the size of the corresponding C data type.

When date SQL data is converted to character C data, the resulting string is in the “yyyy-mm-dd” format. This format is not affected by the MIcrosoft Windows country setting.

SQL to C: Time

The time ODBC SQL data type is:

SQL_TIME

The following table shows the ODBC C data types to which time SQL data may be converted. For an explanation of the columns and terms in the table, see the "Table Description—SQL to C" on page D-23.

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	*BufferLength* > Character byte length 9 <= *BufferLength* <= Character byte length *BufferLength* < 9	Data Truncated data [a] Undefined	Length of data in bytes Length of data in bytes Undefined	N/A 01004 22003
SQL_C_WCHAR	*BufferLength* > Character byte length 9 <= *BufferLength* <= Character byte length *BufferLength* < 9	Data Truncated data [a] Undefined	Length of data in characters Length of data in characters Undefined	N/A 01004 22003
SQL_C_BINARY	Byte length of data <= *BufferLength* > Byte length of data <= *BufferLength*	Data Undefined	Length of data in bytes Undefined	N/A 22003
SQL_C_DATE	None [a]	Data	6 [c]	N/A
SQL_C_TIMESTAMP	None [a]	Data [b]	16 [c]	N/A

[a] The fractional seconds of the time are truncated.
[b] The value of *BufferLength* is ignored for this conversion. The driver assumes that the size of **TargetValuePtr* is the size of the C data type.

[c] The date fields of the timestamp structure are set to the current date and the fractional seconds field of the timestamp structure is set to zero.

[d] This is the size of the corresponding C data type.

When time SQL data is converted to character C data, the resulting string is in the "*hh:mm:ss*" format.

SQL to C: Timestamp

The timestamp ODBC SQL data type is:

SQL_TIMESTAMP

The following table shows the ODBC C data types to which timestamp SQL data may be converted. For an explanation of the columns and terms in the table, see the "Table Description—SQL to C" on page D-23.

C Type Identifier	Test	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	*BufferLength* > Character byte length	Data	Length of data in bytes	N/A
	20 <= *BufferLength* <= Character byte length	Truncated data [b]	Length of data in bytes	01004
	BufferLength < 20	Undefined	Undefined	22003
SQL_C_WCHAR	*BufferLength* > Character byte length	Data	Length of data in characters	N/A
	20 <= *BufferLength* <= Character byte length	Truncated data [b]	Length of data in characters	01004
	BufferLength < 20	Undefined	Undefined	22003
SQL_C_BINARY	Byte length of data <= *BufferLength*	Data	Length of data in bytes	N/A
	Byte length of data > *BufferLength*	Undefined	Undefined	22003

SQL_C_TYPE_DATE	Time portion of timestamp is zero [a]	Data	6 [f]	N/A
	Time portion of timestamp is non-zero [a]	Truncated data [c]	6 [f]	01S07
SQL_C_TYPE_TIME	Fractional seconds portion of timestamp is zero [a]	Data [d]	6 [f]	N/A
	Fractional seconds portion of timestamp is non-zero [a]	Truncated data [d], [e]	6 [f]	01S07
SQL_C_TYPE_TIME STAMP	Fractional seconds portion of timestamp is not truncated [a]	Data [e]	16 [f]	N/A
	Fractional seconds portion of timestamp is truncated [a]	Truncated data [e]	16 [f]	01S07

Notes

[a] The value of *BufferLength* is ignored for this conversion. The driver assumes that the size of **TargetValuePtr* is the size of the C data type.

[b] The fractional seconds of the timestamp are truncated.

[c] The time portion of the timestamp is truncated.

[d] The date portion of the timestamp is ignored.

[e] The fractional seconds portion of the timestamp is truncated.

[f] This is the size of the corresponding C data type.

When timestamp SQL data is converted to character C data, the resulting string is in the "*yyyy-mm-dd hh:mm:ss*[*.f...*]" format, where up to nine digits may be used for fractional seconds. The format is not affected by the Microsoft Windows country setting. (Except for the decimal point and fractional seconds, the entire format must be used, regardless of the precision of the timestamp SQL data type.)

SQL to C Data Conversion Examples

The following examples illustrate how the driver converts SQL data to C data:

SQL Type Identifier	SQL Data Value	C Type Identifier	Buffer Length	*TargetValuePtr	SQLSTATE
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0 [a]	N/A
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0 [a]	N/A
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	4	----	22003
SQL_DECIMAL	1234.56	SQL_C_FLOAT	ignored	1234.56	N/A
SQL_DECIMAL	1234.56	SQL_C_SSHORT	ignored	1234	01S07
SQL_DECIMAL	1234.56	SQL_C_STINYINT	ignored	----	22003
SQL_DOUBLE	1.2345678	SQL_C_DOUBLE	ignored	1.2345678	N/A
SQL_DOUBLE	1.2345678	SQL_C_FLOAT	ignored	1.234567	N/A
SQL_DOUBLE	1.2345678	SQL_C_STINYINT	ignored	1	N/A
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0[a]	N/A
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	10	-----	22003
SQL_TYPE_DATE	1992-12-31	SQL_C_TIMESTAMP	ignored	1992,12,31, 0,0,0,0 [b]	N/A
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 [a]	N/A
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 [a]	01004

SQL_TYPE_ TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	----	22003

[a] "\0" represents a null-termination byte. The driver always null-terminates SQL_C_CHAR data.

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

Converting Data from C to SQL Data Types

When an application calls **SQLExecute** or **SQLExecDirect**, the driver retrieves the data for any parameters bound with **SQLBindParameter** from storage locations in the application. For data-at-execution parameters, the application sends the parameter data with **SQLPutData**. If necessary, the driver converts the data from the data type specified by the *ValueType* argument in **SQLBindParameter** to the data type specified by the *ParameterType* argument in **SQLBindParameter**. Finally, the driver sends the data to the data source.

The following table shows the supported conversions from ODBC C data types to ODBC SQL data types. A solid circle indicates the default conversion for a SQL data type (the C data type from which the data will be converted when the value of *ValueType* or the SQL_DESC_CONCISE_TYPE descriptor field is SQL_C_DEFAULT). A hollow circle indicates a supported conversion.

The format of the converted data is not affected by the Microsoft Windows country setting.

SQL Data Type —SQL_*datatype* where *datatype* is:

C Data Type	CHAR	VARCHAR	LONGVARCHAR	WCHAR	WVARCHAR	WLONGVARCHAR	DECIMAL	NUMERIC	TINYINT (signed)	TINYINT (un-signed)	SMALLINT (signed)	SMALLINT (un-signed)	INTEGER (signed)	INTEGER (un-signed)	BIGINT (signed)	BIGINT (unsigned)	REAL	FLOAT	DOUBLE	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
SQL_C_CHAR	•	•	•	o	o	o	•	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_C_WCHAR	o	o	o	•	•	•	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_C_NUMERIC	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_STINYINT	o	o	o	o	o	o	o	o	•	o	o	o	o	o	o	o	o	o	o						
SQL_C_UTINYINT	o	o	o	o	o	o	o	o	o	•	o	o	o	o	o	o	o	o	o						
SQL_C_TINYINT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_SBIGINT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	•	•	o	o	o						
SQL_C_UBIGINT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	•	o	o	o						
SQL_C_SSHORT	o	o	o	o	o	o	o	o	o	o	•	o	o	o	o	o	o	o	o						
SQL_C_USHORT	o	o	o	o	o	o	o	o	o	o	o	•	o	o	o	o	o	o	o						
SQL_C_SHORT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_SLONG	o	o	o	o	o	o	o	o	o	o	o	o	•	o	o	o	o	o	o						
SQL_C_ULONG	o	o	o	o	o	o	o	o	o	o	o	o	o	•	o	o	o	o	o						
SQL_C_LONG	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_FLOAT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	•	o	o						
SQL_C_DOUBLE	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	•	•						
SQL_C_BINARY	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	•	•	•	o	o	o
SQL_C_DATE	o	o	o	o	o	o																	•		o
SQL_C_TIME	o	o	o	o	o	o																		•	o
SQL_C_TIMESTAMP	o	o	o	o	o	o																	o	o	•

• Default conversion o Supported conversion

Table Description—C to SQL

The tables in the following sections describe how the driver or data source converts data sent to the data source; drivers are required to support conversions from all ODBC C data types to the ODBC SQL data types that they support. For a given ODBC C data type, the first column of the table lists the legal input values of the *ParameterType* argument in **SQLBindParameter**. The second column lists the outcomes of a test that the driver performs to determine if it can convert the data. The third column lists the SQLSTATE returned for each outcome by **SQLExecDirect**, **SQLExecute**, or **SQLPutData**. Data is sent to the data source only if SQL_SUCCESS is returned.

If the *ParameterType* argument in **SQLBindParameter** contains a value for an ODBC SQL data type that is not shown in the table for a given C data type, **SQLBindParameter** returns SQLSTATE 07006 (Restricted data type attribute violation). If the *ParameterType* argument contains a driver-specific value and the driver does not support the conversion from the specific ODBC C data type to that driver-specific SQL data type, **SQLBindParameter** returns SQLSTATE HYC00 (Optional feature not implemented).

If the *ParameterValuePtr* and *StrLen_or_IndPtr* arguments specified in **SQLBindParameter** are both null pointers, that function returns SQLSTATE HY009 (Invalid use of null pointer). Although it is not shown in the tables, an application sets the value pointed to by the *StrLen_or_indPtr* argument of **SQLBindParameter** or the value of the *StrLen_or_indPtr* argument to SQL_NULL_DATA to specify a NULL SQL data value. (The *StrLen_or_indPtr* argument corresponds to the SQL_DESC_OCTET_LENGTH_PTR field of the APD.) The application sets these values to SQL_NTS to specify that the value in **ParameterValuePtr* in **SQLBindParameter** or **DataPtr* in **SQLPutData** (pointed to by the SQL_DESC_DATA_PTR field of the APD) is a null-terminated string.

The following terms are used in the tables:

- **Byte length of data** is the number of bytes of SQL data available to send to the data source, regardless of whether the data will be truncated before it is sent to the data source. For string data, this does not include the null-termination character.
- **Column byte length** is the number of bytes required to store the data at the data source.
- **Character byte length** is the maximum number of bytes needed to display data in character form.
- **Number of digits** is the number of characters used to represent a number, including the minus sign, decimal point, and exponent (if needed).
- Words in *italics* represent elements of the ODBC SQL grammar. See *Appendix C, "SQL Minimum Grammar"* for the syntax of grammar elements.

C to SQL: Character

The character ODBC C data type is:

SQL_C_CHAR
SQL_C_WCHAR

The following table shows the ODBC SQL data types to which C character data may be converted. For an explanation of the columns and terms in the table, see “Table Description—C to SQL” on page D-38.

Note

The length of the Unicode data type must be an even number when character C data is converted to Unicode SQL data.

SQL Type Identifier	Test	SQL-STATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Byte length of data <= Column length	N/A
	Byte length of data > Column length	22001
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Character length of data <= Column length	N/A
	Character length of data > Column length	22001
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT	Data converted without truncation	N/A
	Data converted with truncation of fractional digits [e]	22001
	Conversion of data would result in loss of whole (as opposed to fractional) digits [e]	22001
	Data value is not a *numeric-literal*	22018
SQL_REAL SQL_FLOAT SQL_DOUBLE	Data is within the range of the data type to which the number is being converted	N/A
	Data is outside the range of the data type to which the number is being converted	22003
	Data value is not a *numeric-literal*	22005

SQL_BIT	Data is 0 or 1	N/A
	Data is greater than 0, less than 2, and not equal to 1	22001
	Data is less than 0 or greater than or equal to 2	22003
	Data is not a *numeric-literal*	22018
SQL_BINARY SQL_VARBINARY SQL_LONG-VARBINARY	(Byte length of data) / 2 <= Column byte length	N/A
	(Byte length of data) / 2 > Column byte length	22001
	Data value is not a hexadecimal value	22018
SQL_TYPE_DATE	Data value is a valid *ODBC_date_literal*	N/A
	Data value is a valid *ODBC_timestamp_literal*; time portion is zero	N/A
	Data value is a valid *ODBC_timestamp_literal*; time portion is non-zero [a]	22008
	Data value is not a valid *ODBC_date_literal* or *ODBC_timestamp_literal*	22018
SQL_TYPE_TIME	Data value is a valid *ODBC_time_literal*	N/A
	Data value is a valid *ODBC_timestamp_literal*; fractional seconds portion is zero [b]	N/A
	Data value is a valid *ODBC_timestamp_literal*; fractional seconds portion is non-zero [b]	22008
	Data value is not a valid *ODBC_time_literal* or *ODBC_timestamp_literal*	22018

SQL_TYPE_TIMESTAMP	Data value is a valid *ODBC_timestamp_literal*; fractional seconds portion not truncated	N/A
		22008
	Data value is a valid *ODBC-timestamp-literal*; fractional seconds portion truncated	N/A
		N/A
	Data value is a valid *ODBC-date-literal* [c]	
		22018
	Data value is a valid *ODBC-time-literal* [d]	
	Data value is not a valid *ODBC-date-literal*, *ODBC-time-literal*, or *ODBC-timestamp-literal*	

Notes

[a] The time portion of the timestamp is truncated.

[b] The date portion of the timestamp is ignored.

[c] The time portion of the timestamp is set to zero.

[d] The date portion of the timestamp is set to the current date.

[e] The driver/data source effectively waits until the entire string has been received (even if the character data is sent in pieces by calls to SQLPutData) before attempting to perform the conversion.

When character C data is converted to numeric, date, time, or timestamp SQL data, leading and trailing blanks are ignored.

When character C data is converted to binary SQL data, each two bytes of character data are converted to a single byte (8 bits) of binary data. Each two bytes of character data represent a number in hexadecimal form. For example, “01” is converted to a binary 00000001 and “FF” is converted to a binary 11111111.

The driver always converts pairs of hexadecimal digits to individual bytes and ignores the null termination byte. Because of this, if the length of the character string is odd, the last byte of the string (excluding the null termination byte, if any) is not converted.

Note

Because binding character C data to a binary SQL data type is inefficient and slow, refrain from doing this.

C to SQL: Numeric

The numeric ODBC C data types are:

SQL_C_STINYINT	SQL_C_SLONG
SQL_C_UTINYINT	SQL_C_ULONG
SQL_C_TINYINT	SQL_C_LONG
SQL_C_SSHORT	SQL_C_FLOAT
SQL_C_USHORT	SQL_C_DOUBLE
SQL_C_SHORT	SQL_C_NUMERIC
SQL_C_SBIGINT	SQL_C_UBIGINT

For more information about the SQL_C_TINYINT, SQL_C_SHORT, and SQL_C_LONG data types, see "ODBC 1.0 C Data Types," earlier in this appendix. The following table shows the ODBC SQL data types to which numeric C data may be converted. For an explanation of the columns and terms in the table, see"Table Description—C to SQL" on page D-38.

ParameterType	Test	SQL-STATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Number of digits <= Column byte length Number of digits > Column byte length	N/A 22001
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Number of characters <= Column character length Number of characters > Column character length	N/A 22001
SQL_DECIMAL [a] SQL_NUMERIC [a] SQL_TINYINT [a] SQL_SMALLINT [a] SQL_INTEGER [a] SQL_BIGINT [a]	Data converted without truncation or with truncated of fractional digits Data converted with truncation of whole digits	N/A 22003
SQL_REAL SQL_FLOAT SQL_DOUBLE	Data is within the range of the data type to which the number is being converted Data is outside the range of the data type to which the number is being converted	N/A 22003

Notes

[a] For the "n/a" case, a driver may optionally return SQL_SUCCESS_WITH_INFO and 01S07 when there is a fractional truncation.

The driver ignores the length/indicator value when converting data from the numeric C data types and assumes that the size of the data buffer is the size of the numeric C data type. The length/indicator value is passed in the *StrLen_or_Ind* argument in **SQLPutData** and in the buffer specified with the *StrLen_or_IndPtr* argument in **SQLBindParameter**. The data buffer is specified with the *DataPtr* argument in **SQLPutData** and the *ParameterValuePtr* argument in **SQLBindParameter**.

C to SQL: Bit

The bit ODBC C data type is:

SQL_C_BIT

The following table shows the ODBC SQL data types to which bit C data may be converted. For an explanation of the columns and terms in the table, see “Table Description—C to SQL” on page D-38.

SQL Type Identifier	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	None	N/A
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE	None	N/A

The driver ignores the length/indicator value when converting data from the bit C data types and assumes that the size of the data buffer is the size of the bit C data type. The length/indicator value is passed in the *StrLen_or_Ind* argument in **SQLPutData** and in the buffer specified with the *StrLen_or_IndPtr* argument in **SQLBindParameter**. The data buffer is specified with the *DataPtr* argument in **SQLPutData** and the *ParameterValuePtr* argument in **SQLBindParameter**.

C to SQL: Binary

The binary ODBC C data type is:

SQL_C_BINARY

The following table shows the ODBC SQL data types to which binary C data may be converted. For an explanation of the columns and terms in the table, see "Table Description—C to SQL" on page D-38.

SQL Type Identifier	Test	SQL-STATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Byte length of data <= Column byte length	N/A
	Byte length of data > Column length	22001
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Character length of data <= Column character length	N/A
	Character length of data > Column character length	22001
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE SQL_TYPE_DATE SQL_TYPE_TIME SQL_TYPE_TIMESTAMP	Byte length of data = SQL data length	N/A
	Length of data <> SQL data length	22003
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY	Length of data <= Column length	N/A
	Length of data > Column length	22001

C to SQL: Date

The date ODBC C data type is:

SQL_C_DATE

The following table shows the ODBC SQL data types to which date C data may be converted. For an explanation of the columns and terms in the table, see “Table Description—C to SQL” on page D-38.

ParameterType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column byte length >= 10	N/A
	Column byte length < 10	22001
	Data value is not a valid date	22008
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column character length >= 10	N/A
	Column character length < 10	22001
	Data value is not a valid date	22008
SQL_TYPE_DATE	Data value is a valid date	N/A
	Data value is not a valid date	22007
SQL_TYPE_TIMESTAMP	Data value is a valid date [a]	N/A
	Data value is not a valid date	22007

Notes

[a] The time portion of the timestamp is set to zero.

For information about what values are valid in a SQL_C_TYPE_DATE structure, see “C Data Types” earlier in this appendix.

When date C data is converted to character SQL data, the resulting character data is in the “*yyyy-mm-dd*” format.

The driver ignores the length/indicator value when converting data from the date C data types and assumes that the size of the data buffer is the size of the date C data type. The length/indicator value is passed in the *StrLen_or_Ind* argument in **SQLPutData** and in the buffer specified with the *StrLen_or_IndPtr* argument in **SQLBindParameter**. The data buffer is specified with the *DataPtr* argument in **SQLPutData** and the *ParameterValuePtr* argument in **SQLBindParameter**.

C to SQL: Time

The time ODBC C data type is:

SQL_C_TIME

The following table shows the ODBC SQL data types to which time C data may be converted. For an explanation of the columns and terms in the table, see "Table Description—C to SQL" on page D-38.

ParameterType	Test	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column byte length >= 8	N/A
	Column byte length < 8	22001
	Data value is not a valid time	22008
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Column character length >= 8	N/A
	Column character length < 8	22001
	Data value is not a valid time	22008
SQL_TYPE_TIME	Data value is a valid time	N/A
	Data value is not a valid time	22007
SQL_TYPE_TIMESTAMP	Data value is a valid time [a]	N/A
	Data value is not a valid time	22007

Notes

[a] The date portion of the timestamp is set to the current date and the fractional seconds portion of the timestamp is set to zero.

For information about what values are valid in a SQL_C_TYPE_TIME structure, see "C Data Types" earlier in this appendix.

When time C data is converted to character SQL data, the resulting character data is in the "*hh:mm:ss*" format.

The driver ignores the length/indicator value when converting data from the time C data types and assumes that the size of the data buffer is the size of the time C data type. The length/indicator value is passed in the *StrLen_or_Ind* argument in **SQLPutData** and in the buffer specified with the *StrLen_or_IndPtr* argument in **SQLBindParameter**. The data buffer is specified with the *DataPtr* argument in **SQLPutData** and the *ParameterValuePtr* argument in **SQLBindParameter**.

C to SQL: Timestamp

The timestamp ODBC C data type is:

SQL_C_TIMESTAMP

The following table shows the ODBC SQL data types to which timestamp C data may be converted. For an explanation of the columns and terms in the table, see “Table Description—C to SQL” on page D-38.

SQL Type Identifier	Test	SQL-STATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column byte length >= Character byte length	N/A
	19 <= Column byte length < Character byte length	22001
	Column byte length < 19	22001
	Data value is not a valid date	22008
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	Column character length >= Character length of data	N/A
	19 <= Column character length < Character length of data	22001
	Column character length < 19	22001
	Data value is not a valid timestamp	22008
SQL_TYPE_DATE	Time fields are zero	N/A
	Time fields are non-zero	22008
	Data value does not contain a valid date	22007
SQL_TYPE_TIME	Fractional seconds fields are zero [a]	N/A
	Fractional seconds fields are non-zero [a]	22008
	Data value does not contain a valid time	22007
SQL_TYPE_TIMESTAMP	Fractional seconds fields are not truncated	N/A
	Fractional seconds fields are truncated	22008
	Data value is not a valid timestamp	22007

Notes

[a] The date fields of the timestamp structure are ignored.

For information about what values are valid in a SQL_C_TIMESTAMP structure, see "C Data Types" earlier in this appendix.

When timestamp C data is converted to character SQL data, the resulting character data is in the "*yyyy-mm-dd hh:mm:ss*[.*f*...]" format.

The driver ignores the length/indicator value when converting data from the timestamp C data types and assumes that the size of the data buffer is the size of the timestamp C data type. The length/indicator value is passed in the *StrLen_or_Ind* argument in **SQLPutData** and in the buffer specified with the *StrLen_or_IndPtr* argument in **SQLBindParameter**. The data buffer is specified with the *DataPtr* argument in **SQLPutData** and the *ParameterValuePtr* argument in **SQLBindParameter**.

C to SQL Data Conversion Examples

The following examples illustrate how the driver converts C data to SQL data:

C Data Type	C Data Value	SQL Data Type	Column length	SQL Data Value	SQL-STATE
SQL_C_CHAR	abcdef\0 a	SQL_CHAR	6	abcdef	N/A
SQL_C_CHAR	abcdef\0 a	SQL_CHAR	5	abcde	22001
SQL_C_CHAR	1234.56\0 a	SQL_DECIMAL	8 b	1234.56	N/A
SQL_C_CHAR	1234.56\0 a	SQL_DECIMAL	7 b	1234.5	22001
SQL_C_CHAR	1234.56\0 a	SQL_DECIMAL	4	----	22003
SQL_C_FLOAT	1234.56	SQL_FLOAT	not applicable	1234.56	N/A
SQL_C_FLOAT	1234.56	SQL_INTEGER	not applicable	1234	22001
SQL_C_FLOAT	1234.56	SQL_TINYINT	not applicable	----	22003
SQL_C_TYPE_DATE	1992,12,31 c	SQL_CHAR	10	1992-12-31	N/A
SQL_C_TYPE_DATE	1992,12,31 c	SQL_CHAR	9	----	22003
SQL_C_TYPE_DATE	1992,12,31 c	SQL_TIMESTAMP	not applicable	1992-12-31 00:00:00.0	N/A
SQL_C_TYPE TIMESTAMP	1992,12,31, 23,45,55, 120000000 d	SQL_CHAR	22	1992-12-31 23:45:55.12	N/A

SQL_C_TYPE TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	21	1992-12-31 23:45:55.1	22001
SQL_C_TYPE TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	18	----	22003

Notes

[a] "\0" represents a null-termination byte. The null-termination byte is required only if the length of the data is SQL_NTS.

[b] In addition to bytes for numbers, one byte is required for a sign and another byte is required for the decimal point.

[c] The numbers in this list are the numbers stored in the fields of the SQL_DATE_STRUCT structure.

[d] The numbers in this list are the numbers stored in the fields of the SQL_TIMESTAMP_STRUCT structure.

E

Scalar Functions

ODBC specifies five types of scalar functions:

- String functions
- Numeric functions
- Time and date functions
- System functions
- Data type conversion functions

This appendix includes tables for each scalar function category. Within each table, functions have been added in ODBC 3.0 to align with SQL-92. Each table also provides the version number when the function was introduced.

ODBC and SQL-92 Scalar Functions

Because functions are often data-source-specific, ODBC does not require a data type for return values from scalar functions. To force data type conversion, applications should use the CONVERT scalar function.

Note

Keep in mind the different ways in which ODBC and SQL-92 classify functions. ODBC classifies scalar functions by argument type, whereas SQL-92 classifies them by return value. For example, the EXTRACT function is an ODBC timedate function because the extract-field argument is a datetime keyword and the extract_source argument is a datetime or interval expression. In SQL-92, the EXTRACT function is a numeric scalar function because the return value is numeric.

Applications need to call **SQLGetInfo** to determine which scalar functions a driver supports. ODBC and SQL-92 information types are available for scalar function classifications. Because ODBC and SQL-92 use different classfications, the information types for the same function may differ between ODBC and SQL-92. For example, to determine support for the EXTRACT function requires SQL_TIMEDATE_FUNCTIONS information type in ODBC and SQL_SQL92_NUMERIC_VALUE_FUNCTIONS information type in SQL-92.

String Functions

This section lists string manipulation functions. Applications can call **SQLGetInfo** with the SQL_STRING_FUNCTIONS information type to determine which string functions are supported by a driver.

String Function Arguments

Arguments denoted as...	Definition
string_exp	can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, or SQL_LONGVARCHAR.
start, *length* or *count*	can be a numeric literal or the result of another scalar function, where the underlying data type can be represented as SQL_TINYINT, SQL_SMALLINT, or SQL_INTEGER
character_exp	are a variable-length character string

The following string functions are 1-based, that is, the first character in the string is character 1

Note

BIT_LENGTH, CHAR_LENGTH, CHARACTER_LENGTH, OCTET_LENGH, and POSITION string scalar functions were added in ODBC 3.0 to align with SQL-92.

List of String Functions

Function	Description
ASCII(*string_exp*) (ODBC 1.0)	Returns the ASCII code value of the leftmost character of *string_exp* as an integer.
BIT_LENGTH(*string_exp*) (ODBC 3.0)	Retruns the length in bits of string expression.
CHAR(*code*) (ODBC 1.0)	Returns the character that has the ASCII code value specified by code. The value of *code* should be between 0 and 255; otherwise, the return value is data source–dependent.
CHAR_LENGTH(*string_exp*) (ODBC 3.0)	Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer notless than the number of bits divided by 8). (This function is the same as CHARACTER_LENGTH function.)
CHARACTER_LENGTH(*string_exp*) (ODBC 3.0)	Returns the length in characters of the string expression, if the string expression is of a character data type; otherwise, returns the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHAR_LENGTH function.)
CONCAT(*string_exp1*, *string_exp2*) (ODBC 1.0)	Returns a character string that is the result of concatenating *string_exp2* to *string_exp1*. The resulting string is DBMS-dependent.
INSERT(*string_exp1*, *start*, *length*, *string_exp2*) (ODBC 1.0)	Returns a character string where *length* characters have been deleted from *string_exp1* beginning at *start* and where *string_exp2* has been inserted into *string_exp*, beginning at start.
LCASE(*string_exp*) (ODBC 1.0)	Returns a string equal to that *string_exp*, with all uppercase characters converted to lowercase..
LEFT(*string_exp*, *count*) (ODBC 1.0)	Returns the leftmost *count* of characters of *string_exp*.
LENGTH(*string_exp*) (ODBC 1.0)	Returns the number of characters in *string_exp,* excluding trailing blanks.

LOCATE(*string_exp1*, *string_exp2*[, *start*])	Returns the starting position of the first occurrence of *string_exp1* within *string_exp2*. The search for the first occurrence of *string_exp1* begins with the first character position in *string_exp2* unless the optional argument, *start,* is specified. If *start* is specified, the search begins with the character position indicated by the value of *start*. The first character position in *string_exp2* is indicated by the value 1. If *string_exp1* is not found within *string_exp2*, the value 0 is returned. If an application can call the LOCATE scalar function with the *string_exp1*, *string_exp2,* and *start* arguments, the driver returns SQL_FN_STR_LOCATE when **SQLGetInfo** is called with an *Option* of SQL_STRING_FUNCTIONS. If the application can call the LOCATE scalar function with only the *string_exp1* and *string_exp2* arguments, the driver returns SQL_FN_STR_LOCATE_2 when SQLGetInfo is called with an *Option* of SQL_STRING_FUNCTIONS. Drivers that support calling the LOCATE function with either two or three arguments return both SQL_FN_STR_LOCATE and SQL_FN_STR_LOCATE_2.
LTRIM(*string_exp*) (ODBC 1.0)	Returns the characters of *string_exp*, with leading blanks removed.
OCTET_LENGTH(*string_exp*) (ODBC 3.0)	Returns the length in bytes of the string expression. The result is the smallest integer not less than the number of bits divided by 8.
POSITION(*character_exp* IN *character_exp*) (ODBC 3.0)	Returns the position of the first character expression in the second character expression. The result is an exact numeric with an implementation-defined precison and a scale of 0.
REPEAT(*string_exp*, *count*) (ODBC 1.0)	Returns a character string composed of *string_exp* repeated *count* times.
REPLACE(*string_exp1*, *string_exp2*, *string_exp3*) (ODBC 1.0)	Search *string_exp1* for occurrences of *string_exp2*, and replace with *string_exp3*.

RIGHT(*string_exp*, *count*) (ODBC 1.0)	Returns the rightmost *count* of characters of *string_exp*.
RTRIM(*string_exp*) (ODBC 1.0)	Returns the characters of *string_exp* with trailing blanks removed.
SPACE(*count*) (ODBC 2.0)	Returns a character string consisting of *count* spaces.
SUBSTRING(*string_exp*, *start*, *length*) (ODBC 1.0)	Returns a character string that is derived from *string_exp*, beginning at the character position specified by *start* for *length* characters.
UCASE(*string_exp*) (ODBC 1.0)	Returns a string equal to that in *string_exp*, with all lowercase characters converted to uppercase.

Numeric Functions

This section describes numeric functions that are included in the ODBC scalar function set. Applications can call **SQLGetInfo** with the SQL_NUMERIC_FUNCTIONS information type to determine which string functions are supported by a driver.

Except for ABS, ROUND, TRUNCATE, SIGN, FLOOR, and CEILING (which return values of the same data type as the input parameters), all numeric functions return values of data type SQL_FLOAT.

Numberic Function Arguments

Arguments denoted as...	Definition
numeric_exp	can be the name of a column, the result of another scalar function, or a numeric literal, where the underlying data type could be represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, or SQL_DOUBLE
float_exp	can be the name of a column, the result of another scalar function, or a numeric literal, where the underlying data type can be represented as SQL_FLOAT.
integer_exp	can be the name of a column, the result of another scalar function, or a numeric literal, where the underlying data type can be represented as SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, or SQL_BIGINT

List of Numeric Functions

Function	Description
ABS(*numeric_exp***)** (ODBC 1.0)	Returns the absolute value of *numeric_exp*.
ACOS(*float_exp***)** (ODBC 1.0)	Returns the arccosine of *float_exp* as an angle, expressed in radians.
ASIN(*float_exp***)** (ODBC 1.0)	Returns the arcsine of *float_exp* as an angle, expressed in radians.
ATAN(*float_exp***)** (ODBC 1.0)	Returns the arctangent of *float_exp* as an angle, expressed in radians.
ATAN2(*float_exp1***,** *float_exp2***)** (ODBC 2.0)	Returns the arctangent of the x and y coordinates, specified by *float_exp1* and *float_exp2*, respectively, as an angle, expressed in radians.
CEILING(*numeric_exp***)** (ODBC 1.0)	Returns the smallest integer greater than or equal to *numeric_exp*. The return value is of the same data type as the input parameter.
COS(*float_exp***)** (ODBC 1.0)	Returns the cosine of *float_exp*, where *float_exp* is an angle expressed in radians.
COT(*float_exp***)** (ODBC 1.0)	Returns the cotangent of *float_exp*, where *float_exp* is an angle expressed in radians.
DEGREES(*numeric_exp***)** (ODBC 2.0)	Returns the number of degrees converted from *numeric_exp* radians.
EXP(*float_exp***)** (ODBC 1.0)	Returns the exponential value of *float_exp*.
FLOOR(*numeric_exp***)** (ODBC 1.0)	Returns largest integer less than or equal to *numeric_exp*. The return value is of the same data type as the input parameter.
LOG(*float_exp***)** (ODBC 1.0)	Returns the natural logarithm of *float_exp*.

LOG10(*float_exp***)** (ODBC 2.0)	Returns the base 10 logarithm of *float_exp*.
MOD(*integer_exp1***,** *integer_exp2***)** (ODBC 1.0)	Returns the remainder (modulus) of *integer_exp1* divided by *integer_exp2*.
PI( ) (ODBC 1.0)	Returns the constant value of pi as a floating point value.
POWER(*numeric_exp***,** *integer_exp***)**	Returns the value of *numeric_exp* to the power of *integer_exp*.
RADIANS(*numeric_exp***)** (ODBC 2.0)	Returns the number of radians converted from *numeric_exp* degrees.
RAND([*integer_exp*]) (ODBC 1.0)	Returns a random floating-point value using *integer_exp* as the optional seed value.
ROUND(*numeric_exp***,** *integer_exp***)** (ODBC 2.0)	Returns *numeric_exp* rounded to *integer_exp* places right of the decimal point. If *integer_exp* is negative, *numeric_exp* is rounded to |*integer_exp*| places to the left of the decimal point.
SIGN(*numeric_exp***)** (ODBC 1.0)	Returns an indicator or the sign of *numeric_exp*. If *numeric_exp* is less than zero, –1 is returned. If *numeric_exp* equals zero, 0 is returned. If *numeric_exp* is greater than zero, 1 is returned.
SIN(*float_exp***)** (ODBC 1.0)	Returns the sine of *float_exp,* where f*loat_exp* is an angle expressed in radians.
SQRT(*float_exp***)** (ODBC 1.0)	Returns the square root of *float_exp*.
TAN(*float_exp***)** (ODBC 1.0)	Returns the tangent of *float_exp,* where *float_exp* is an angle expressed in radians.
TRUNCATE(*numeric_exp***,** *integer_exp***)** (ODBC 2.0)	Returns *numeric_exp* truncated to *integer_exp* places right of the decimal point. If *integer_exp* is negative, *numeric_exp* is truncated to |*integer_exp*| places to the left of the decimal point.

Time and Date Functions

This section lists time and date functions that are included in the ODBC scalar function set. Applications can call **SQLGetInfo** with the SQL_TIMEDATE_FUNCTIONS information type to determine which time and date functions are supported by a driver.

Time and Data Arguments

Arguments denoted as...	Definition
timestamp_exp	can be the name of a column, the result of another scalar function, or an *ODBC_time_escape*, *ODBC_date_escape*, or *ODBC_timestamp_escape*, where the underlying data type could be represented as SQL_CHAR, SQL_VARCHAR, SQL_TYPE_TIME, SQL_TYPE_DATE, or SQL_TYPE_TIMESTAMP.
date_exp	can be the name of a column, the result of another scalar function, or an *ODBC_date_escape* or *ODBC_timestamp_escape*, where the underlying data type could be represented as SQL_CHAR, SQL_VARCHAR, SQL_TYPE_DATE, or SQL_TYPE_TIMESTAMP.
time_exp	can be the name of a column, the result of another scalar function, or an *ODBC_time_escape* or *ODBC_timestamp_escape*, where the underlying data type could be represented as SQL_CHAR, SQL_VARCHAR, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP

Note

CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP timedate scalar functions were added in ODBC 3.0 to align with SQL-92.

List of Time and Date Functions

Function	Description
CURRENTTIME[(*time_precision*)] (ODBC 3.0)	Returns the current local time as a time value. The *time_precision* argument determines the seconds precision of the returned value.
CURRENT_TIMESTAMP[(*timestamp_precision*)] (ODBC 3.0)	Returns the current local data and local time as a timestamp value. The *timestamp_precision* argument determines the seconds precision of the returned timestamp.
CURDATE( ) (ODBC 1.0)	Returns the current date.
CURTIME( ) (ODBC 1.0)	Returns the current local time.
DAYNAME(*date_exp***)** (ODBC 2.0)	Returns a character string containing the data source–specific name of the day (for example, Sunday, through Saturday or Sun. through Sat. for a data source that uses English, or Sonntag through Samstag for a data source that uses German) for the day portion of *date_exp*.
DAYOFMONTH(*date_exp***)** (ODBC 1.0)	Returns the day of the month in *date_exp* as an integer value in the range of 1–31.
DAYOFWEEK(*date_exp***)** (ODBC 1.0)	Returns the day of the week based on the week field in *date_exp* as an integer value in the range of 1–7, where 1 represents Sunday.
DAYOFYEAR(*date_exp***)** (ODBC 1.0)	Returns the day of the year based on the year field in *date_exp* as an integer value in the range of 1–366.

EXTRACT(*extract_field* FROM *extract_source*) (ODBC 3.0)	Returns the *extract_field* portion of the *extract_source*. The *extract_source* argument is a datetime or interval xpression. The *extract_field* argument can be one of the following keywords" YEAR MONTH DAY HOUR MINUTE SECOND The precision of the returned value is implementation-defined. The scale is 0 unless SECOND is specified, in which case the scale is not less than the fractional seconds precision of the *extract_source* field.
HOUR(*time_exp***)** (ODBC 1.0)	Returns the hour based on the hour field in *time_exp* as an integer value in the range of 0–23.
MINUTE(*time_exp***)** (ODBC 1.0)	Returns the minute based on the minute field in *time_exp* as an integer value in the range of 0–59.
MONTH(*date_exp***)** (ODBC 1.0)	Returns the month based on the month field in *date_exp* as an integer value in the range of 1–12.
MONTHNAME(*date_exp***)** (ODBC 2.0)	Returns a character string containing the data source–specific name of the month (for example, January through December or Jan. through Dec. for a data source that uses English, or Januar through Dezember for a data source that uses German) for the month portion of *date_exp*.
NOW( ) (ODBC 1.0)	Returns current date and time as a timestamp value.
QUARTER(*date_exp***)** (ODBC 1.0)	Returns the quarter in *date_exp* as an integer value in the range of 1–4, where 1 represents January 1 through March 31.
SECOND(*time_exp***)** (ODBC 1.0)	Returns the second in *time_exp* as an integer value in the range of 0-59.

TIMESTAMPADD(*interval***,** *integer_exp***,** *timestamp_exp***)** (ODBC 2.0)

Returns the timestamp calculated by adding *integer_exp* intervals of type *interval* to *timestamp_exp*. Valid values of interval are the following keywords:

SQL_TSI_FRAC_SECOND
SQL_TSI_FRAC_SECOND
SQL_TSI_MINUTE
SQL_TSI_HOUR
SQL_TSI_DAY
SQL_TSI_WEEK
SQL_TSI_MONTH
SQL_TSI_QUARTER
SQL_TSI_YEAR

where fractional seconds are expressed in billionths of a second For example, the following SQL statement returns the name of each emplyee and his or her one-year anniversary date:

```
SELECT NAME, {fn
TIMESTAMPADD(SQL_TSI_YEAR, 1,
HIRE_DATE)} FROM
EMPLOYEES
```

If *timestamp_exp* is a time value and interval specfies day, weeks, months, quarters, or years, the date portion of *timestamp_exp* is set to the current date before calculating the resulting timestamp.

If *timestamp_exp* is a date value and interval specifies fractional seconds, seconds, minutes, or hours, the time portion of *timestamp_exp* is set to 0 before calculating the resulting timestamp.

An application determines which intervals a data source supports by calling **SQLGetInfo** with the SQL_TIMEDATE_ADD_INTERVALS option.

TIMESTAMPDIFF(*interval***,** *timestamp_exp1***,** *timestamp_exp2***)** (ODBC 2.0)

Returns the integer number of intervals of type *interval* as the amount of full units between *timestamp_exp1* and *timestamp_exp2*.

If an application relies on the old TIMESTAMPDIFF semantics, the old behavior can be emulated by the following configuration setting in the SQL section of the solid.ini file.

```
[SQL]
EmulateOLdTIMESTAMPDIFF=YES
```

Note that the old semantics returns the integer number of intervals of type *interval* by which *timestamp_exp2* is greater than *timestamp_exp1*.

Valid values of *interval* are the following keywords:

SQL_TSI_FRAC_SECOND
SQL_TSI_SECOND
SQL_TSI_MINUTE
SQL_TSI_HOUR
SQL_TSI_DAY
SQL_TSI_WEEK
SQL_TSI_MONTH
SQL_TSI_QUARTER
SQL_TSI_YEAR

where fractional seconds are expressed in billionths of a second. For example, the following SQL statement returns the name of each employee and the number of years they have been employed:

TIMESTAMPDIFF(*interval***,** *timestamp_exp1***,** *timestamp_exp2***) (continued)**	SELECT NAME, {fn TIMESTAMPDIFF(SQL_TSI_YEAR, {fn CURDATE()}, HIRE_DATE)} FROM EMPLOYEES If either timestamp expression is a time value and *interval* specifies days, weeks, months, quarters, or years, the date portion of that timestamp is set to the current date before calculating the difference between the timestamps. If either timestamp expression is a date value and *interval* specifies fractional seconds, seconds, minutes, or hours, the time portion of of that timestamp is set to 0 before calculating the difference between the timestamps. An application determines which intervals a data source supports by calling **SQLGetInfo** with the SQL_TIMEDATE_DIFF_INTERVALS option.
WEEK(*date_exp***)** (ODBC 1.0)	Returns the week of the year based on the week field in *date_exp* as an integer value in the range of 1–53.
YEAR(*date_exp***)** (ODBC 1.0)	Returns the year based on the year field in *date_exp* as an integer value. The range is data source–dependent.

System Functions

This section lists system functions that are included in the ODBC scalar function set. Applications can call **SQLGetInfo** with the SQL_SYSTEM_FUNCTIONS information type to determine which string functions are supported by a driver.

System Functions Arguments

Arguments denoted as...	Definition
exp	can be the name of a column, the result of another scalar function, or a literal, where the underlying data type could be represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, SQL_DOUBLE, SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP.

Arguments denoted as...	Definition
value	can be a literal constant, where the underlying data type can be represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, SQL_DOUBLE, SQL_TYPE_DATE, SQL_TYPE_TIME, or SQL_TYPE_TIMESTAMP.
integer_exp	can be the name of a column, the result of another scalar function, or a numeric literal, where the underlying data type can be represented as SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, or SQL_BIGINT

Values returned are represented as ODBC data types

List of System Functions

.

Function	Description
DATABASE( ) (ODBC 1.0)	Returns the name of the database corresponding to the connection handle. (The name of the database is also available by calling **SQLGetConnectOption** with the SQL_CURRENT_QUALIFIER connection option.)
IFNULL(*exp***,***value* (ODBC 1.0)	If *exp* is null, *value* is returned. If *exp* is not null, *exp* is returned. The possible data type(s) of *value* must be compatible with the data type of *exp*
USER( ) (ODBC 1.0)	Returns the user's name in the DBMS. (The user's authorization name is also available via **SQLGetInfo** by specifying the information type: SQL_USER_NAME.) This can be different from the login time.

Explicit Data Type Conversion

Explicit data type conversion is specified in terms of SQL data type definitions.

The ODBC syntax for the explicit data type conversion function does not restrict conversions. The validity of specific conversions of one data type to another data type is dependent on each driver-specific implementation. The driver, as it translates the ODBC syntax into the native syntax, reject those conversions that, although legal in the ODBC syntax, are not supported by the data source. Applications can call the ODBC function **SQLGetInfo** to inquire about conversions supported by the data source.

The format of the **CONVERT** function is:

CONVERT(*value_exp***,** *data_type***)**

The function returns the value specified by *value_exp* converted to the specified *data_type*, where *data_type* is one of the following keywords:

SQL_BIGINT	SQL_SMALLINT
SQL_BINARY	SQL_DATE
SQL_CHAR	SQL_TIME
SQL_DECIMAL	SQL_TIMESTAMP
SQL_DOUBLE	SQL_TINYINT
SQL_FLOAT	SQL_VARBINARY
SQL_INTEGER	SQL_VARCHAR
SQL_LONGVARBINARY	SQL_WCHAR
SQL_LONGVARCHAR	SQL_WLONGVARCHAR
SQL_NUMERIC	SQL_WVARCHAR
SQL_REAL	

The ODBC syntax for the explicit data type conversion function does not support specification of conversion format. If specification of explicit formats is supported by the underlying data source, a driver must specify a default value or implement format specification.

The argument *value_exp* can be a column name, the result of another scalar function, or a numeric or string literal. For example:

{ fn CONVERT({ fn CURDATE() }, SQL_CHAR) }

converts the output of the CURDATE scalar function to a character string.

ODBC does not require a data type for return values from scalar functions (because the functions are often data source-specific); applications should use the CONVERT scalar function whenever possible to force data type conversion.

The following two examples illustrate the use of the **CONVERT** function. These examples assume the existence of a table called EMPLOYEES, with an EMPNO column of type SQL_SMALLINT and an EMPNAME column of type SQL_CHAR.

If an application specifies the following:

```
SELECT EMPNO FROM EMPLOYEES WHERE {fn CONVERT(EMPNO,SQL_CHAR)}LIKE '1%'
```

SOLID ODBC driver translates the request to:

```
SELECT EMPNO FROM EMPLOYEES WHERE CONVERT_CHAR(EMPNO) LIKE '1%'
```

If an application specifies the following:

```
SELECT {fn ABS(EMPNO)}, {fn CONVERT(EMPNAME,SQL_SMALLINT)}
    FROM EMPLOYEES WHERE EMPNO <> 0
```

SOLID ODBC driver translates the request to:

```
SELECT ABS(EMPNO), CONVERT_SMALLINT(EMPNAME) FROM EMPLOYEES
    WHERE EMPNO <> 0
```

SQL-92 CAST Function

The ODBC CONVERT function has an equivalent function in SQL-92: the CAST function. The syntax for these equivalent functions are:

```
{ fn CONVERT (value_exp, data_type)} / * ODBC
CAST (value_exp AS data_type) /* SQL 92
```

Support for the CAST function is at the FIPS Transitional level. For details on data type conversion in the CAST function, see the SQL-92 specification.

To determine application support for the CAST function, call **SQLGetInfo** with the SQL_SQL_CONFORMANCE information type. The CAST function is supported if the return value for the information type is:

- SQL_SC_FIPS127_2_TRANSITIONAL
- SQL_SC_SQL92_INTERMEDIATE
- SQL_SC_SQL92_FULL

If the return value is SQL_SC_ENTRY or 0, call **SQLGetInfo** with the SQL_SQL92_VALUE_EXPRESSIONS information type. If the SQL_SVE_CAST bit is set, the CAST function is supported.

Index

D

E

F

G

I