LabWindows®/CVI
SQL Toolkit Reference Manual

May 1995 Edition

Part Number 320960A-01

National Instruments Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
Technical support fax: (800) 328-2203
(512) 794-5678

Branch Offices:
Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Canada (Ontario) 519 622 9310,
Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 90 527 2321, France 1 48 14 24 24,
Germany 089 741 31 30, Hong Kong 2645 3186, Italy 02 48301892, Japan 03 5472 2970, Korea 02 596 7456,
Mexico 5 202 2544, Netherlands 03480 33466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085,
Sweden 08 730 49 70, Switzerland 056 20 51 51, Taiwan 02 377 1200, U.K. 01635 523545

Limited Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

Product and company names listed are trademarks or trade names of their respective companies.

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

Chapter 4
SQL Toolkit Function Descriptions 4-1

Figures

Tables

About This Manual

The *LabWindows/CVI SQL Toolkit Reference Manual* describes the LabWindows/CVI add-on package you can use for database operations.

Organization of This Manual

The *LabWindows/CVI SQL Toolkit Reference Manual* is organized as follows.

- Chapter 1, *Introduction*, describes the installation procedure and lists the main features of the SQL Toolkit.
- Chapter 2, *Getting Started*, introduces the basic concepts of database interactions using the LabWindows/CVI SQL Toolkit. It also describes the Structured Query Language (SQL), the ODBC Standard, and the Database Session.
- Chapter 3, *Using the SQL Toolkit*, describes how to use the SQL Toolkit functions for common types of database operations and contains example code for performing each operation.
- Chapter 4, *SQL Toolkit Function Descriptions*, describes the functions in the LabWindows/CVI SQL Toolkit. The *SQL Toolkit Overview* section contains general information about the SQL Toolkit functions. The *SQL Toolkit Function Reference* section contains an alphabetical list of function descriptions.
- Appendix A, *SQL Reference*, briefly explains SQL commands, operators, and functions. This version of SQL is included in the ODBC standard and applies to all ODBC-compliant databases.
- Appendix B, *Error Codes*, describes the error codes returned by functions in the LabWindows/CVI SQL Toolkit. In many cases, you can obtain additional information about errors by using `DBErrorMessage`.
- Appendix C, *Format Strings*, describes the format strings that you can use with `DBMapColumnToChar` and `DBBindColChar`.
- Appendix D, *Customer Communication*, contains forms to help you gather the information necessary to help National Instruments solve technical problems you might have as well as a form you can use to comment on our products and manuals.

Conventions Used in This Manual

The following conventions are used in this manual:

bold	Bold text denotes menus, menu items, or dialog box buttons or options.
italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
bold italic	Bold, italic text denotes a note, caution, or warning.
`monospace`	Lowercase text in this font denotes text or characters that you should literally enter from the keyboard. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, variables, filenames, and extensions, and for statements and comments taken from program code.
`italic monospace`	Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.
<>	Angle brackets enclose the name of a key on the keyboard— for example, <PageDown>. A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Control-Alt-Delete>.
<Control>	Key names are capitalized.

Related Documentation

The following documents contain information that you may find helpful as you read this manual:

- *Getting Started with LabWindows/CVI*
- *LabWindows/CVI User Manual*

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. You will find these forms in Appendix D, *Customer Communication*, at the end of this manual.

Chapter 1
Introduction

This chapter describes the installation procedure and lists the main features of the SQL Toolkit.

Installation

This section contains instructions for installing the LabWindows/CVI SQL Toolkit on the Windows platform.

Installing the SQL Toolkit Software

Insert LabWindows/CVI SQL Toolkit disk one into the 3.5 in. disk drive and run the SETUP.EXE program using one of the following methods.

- From Windows, select **Run...** from the **File** menu of the Program Manager. In the dialog box that appears, type X:\SETUP (where X denotes the proper drive designation).
- From Windows, launch the File Manager. Double-click on the drive icon that contains the installation disk. Find SETUP.EXE in the list of files on that disk and double-click on it.

Figure 1-1 shows the LabWindows/CVI SQL Toolkit installation dialog box.

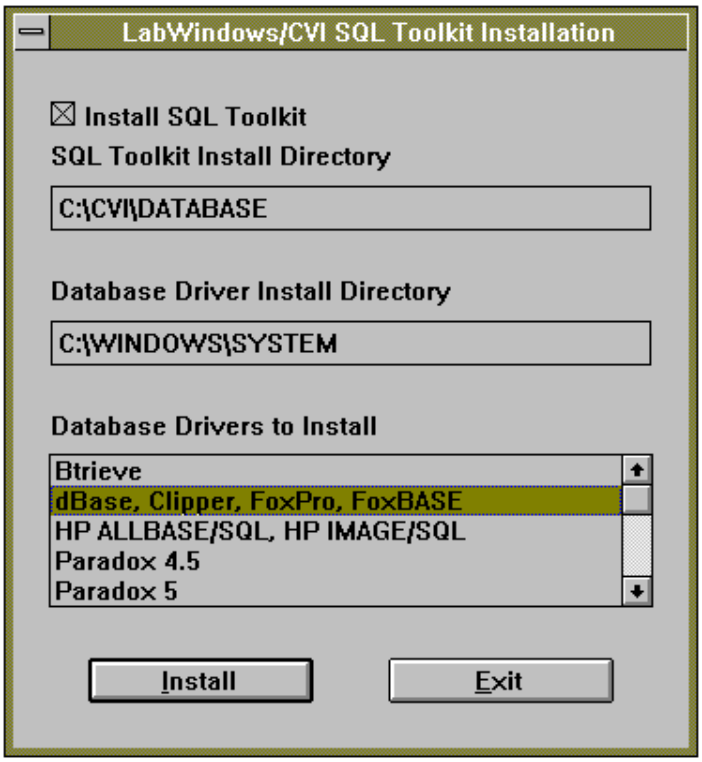

Figure 1-1. LabWindows/CVI SQL Toolkit Installation Dialog Box

When the installation dialog box appears on the screen, you can change the default directories for the toolkit and the ODBC drivers. The LabWindows/CVI SQL Toolkit installation program contains the following two groups of installation files.

- SQL Toolkit files. Contains library file function panels and LabWindows/CVI example programs for database communication. The default installation directory for the SQL Toolkit files is the `\DATABASE` subdirectory within the LabWindows/CVI base directory

- ODBC Database Driver files. Contains database-specific DLLs required to communicate with each database. The ODBC Database Driver files default installation directory is the `\WINDOWS\SYSTEM` directory.

The installation program also installs the following files in the Windows system directory.

- `ODBC.DLL`
- `ODBCINST.DLL`
- `ODBCCURS.DLL`
- `CTL3DV2.DLL`.

In addition, the driver setup program modifies or creates `CONTROL.INI`, `ODBC.INI`, and `ODBCINST.INI` in the Windows directory.

If you decide to install the ODBC Database Driver files into a directory other than a directory specified in your current PATH statement, you must add your new driver directory to your PATH statement in your `AUTOEXEC.BAT` file. You must then reboot your computer for the new path to take effect.

Note: ***The LabWindows/CVI SQL Toolkit ODBC Database Driver files consist of Closed Drivers, in that, they can only be used by LabWindows/CVI. The installation program will install two driver files,*** `QECD.LIC` ***and*** `QELIB.DLL`***, that enable the LabWindows/CVI SQL Toolkit to use the supplied Closed Drivers. Some older versions of Microsoft Excel and some other database products may also use their own versions of these driver files. This may prevent the LabWindows/CVI SQL Toolkit from functioning properly. If an older*** `QELIB` ***file exists on your system and it occurs earlier within the defined search path for DLLs, you may have to remove it or place the ODBC Database Driver files earlier with your PATH statement.***

Overview

The LabWindows/CVI SQL Toolkit is an add-on package for accessing databases. The toolkit contains a set of high-level functions for performing the most common database tasks.

The following list describes the main features of the LabWindows/CVI SQL Toolkit.

- Works with any database driver that complies with Microsoft's Open Database Connectivity (ODBC) standard.
- Maintains a high level of portability. In many cases, you can port your application to another database by simply changing the connection string passed to the `DBConnect` function.
- Converts database column values from native data types to eight, standard SQL Toolkit data types, further enhancing portability.
- Permits the use of SQL statements with all supported database systems, even non-SQL systems.
- Includes functions to retrieve the name and data type of a column returned by a `SELECT` statement.
- Scrolls backward and forward through records returned by a `SELECT` statement, even in databases that do not support backward scrolling.
- Creates tables and selects, inserts, updates, and deletes records without using SQL statements.
- Allows the use of transactions, which group database operations together so that they can be committed or canceled as a unit, with databases that do not support transactions.

Because of the wide range of databases the SQL Toolkit works with, some portability issues remain. You should consider the following when choosing your database system.

- Some database systems, particularly the flat-file databases such as dBase, do not support floating point numbers. In such cases, the SQL Toolkit converts floating point numbers to the nearest equivalent, usually Binary Coded Decimal (BCD), before storing them in the database. Very large or very small floating point numbers can easily overflow or underflow the precision available for a BCD.

- Restrictions on column names vary among database systems. For maximum portability, limit column names to 10 uppercase characters without embedded spaces.

- Some database systems do not support date, time, or date and time data types.

Chapter 2
Getting Started

This chapter introduces the basic concepts of database interactions using the LabWindows/CVI SQL Toolkit. It also describes the Structured Query Language (SQL), the Open Database Connectivity (ODBC) Standard, and the Database Session.

Database Concepts

A database consists of an organized collection of data. While the underlying details may vary, most modern Database Management Systems (DBMS) store data in tables. The tables are organized into rows, also known as records, and columns, also known as fields. Every table in a database must have a unique name. Similarly, every column within a table must have a unique name.

The database tables have many uses. Table 2-1 is an example table that you could use with a simple test executive program to record test sequence results. It contains columns for the Unit Under Test (UUT) number, the test name, the test result, and two measurements.

Table 2-1. Sample Test Sequence Results

UUT_NUM	TEST_NAME	RESULT	MEAS1	MEAS2
20860B456	TEST1	PASS	0.5	0.6
20860B456	TEST2	PASS	1.2	
20860B123	TEST1	FAIL	-0.1	0.7
20860B789	TEST1	PASS	0.6	0.6
20860B789	TEST2	PASS	1.3	

The data in the table are not inherently ordered. Ordering, grouping, and other manipulations of the data occur when you use a `SELECT` statement to retrieve the data from the table. A row can have empty columns, which means that the row contains `NULL` values. Notice that the `NULL` values in a table row are not the same as `NULL` values in the C programming language.

Each column in a table has a data type. The available data types vary depending on the DBMS. The LabWindows/CVI SQL Toolkit uses a set of common data types. The SQL Toolkit automatically maps these data types into the appropriate type in the underlying database. By using the common data types, the SQL Toolkit program can access a variety of databases with little or no modification.

Table 2-2 lists the data types supported by the SQL Toolkit.

Table 2-2. Data Types Supported by the SQL Toolkit

Type Code	Type Constant Name	Data Type Description
1	DB_CHAR	Fixed-length character string.
2	DB_VARCHAR	Character string.
3	DB_DECIMAL	Binary Coded Decimal (BCD).
4	DB_INTEGER	Long integer.
5	DB_SMALLINT	Short integer.
6	DB_FLOAT	Single-precision floating point.
7	DB_DOUBLEPRECISION	Double-precision floating point.
8	DB_DATETIME	Date/time. (YYYY-MM-DD HH:MM:SS.SSSSSS).

The Structured Query Language (SQL)

The Structured Query Language (SQL) consists of a widely supported standard for database access. You can use the SQL commands to manipulate the rows and columns in database tables. The following list describes some of the most useful SQL commands.

- CREATE TABLE—Creates a new table specifying the name and data type for each column.
- SELECT—Retrieves all rows in a table that match specified conditions.
- INSERT—Adds a new record to the table. You can then assign values for the columns.
- UPDATE—Changes values in specified columns for all rows that match specified conditions.
- DELETE—Deletes all rows that match specified conditions.

See Appendix A, *SQL Reference*, for a complete list of SQL commands.

The ODBC Standard

Microsoft Corp. developed the Open Database Connectivity (ODBC) Standard as a uniform method for applications to access databases. The standard includes a driver packaging standard, a method for maintaining Data Source Names, and a SQL implementation based on ANSI SQL. Because the LabWindows/CVI SQL Toolkit and the supplied drivers comply with the ODBC standard, you can port LabWindows/CVI database applications to other supported databases with minimal changes.

The ODBC Administrator

On Microsoft Windows 3.*x* systems, ODBC drivers consist of 16-bit DLL drivers. Any ODBC driver that you use must be registered into the `ODBC.INI` text file located in the Windows directory. You can use the ODBC Administrator icon on your Control Panel to configure a driver to make it available as a data source for your applications. Your system will save all changes made within the ODBC Administrator into the `ODBC.INI` file.

By double-clicking on the ODBC Administrator icon on the Control Panel, the Data Sources dialog box, located in Figure 2-1, appears.

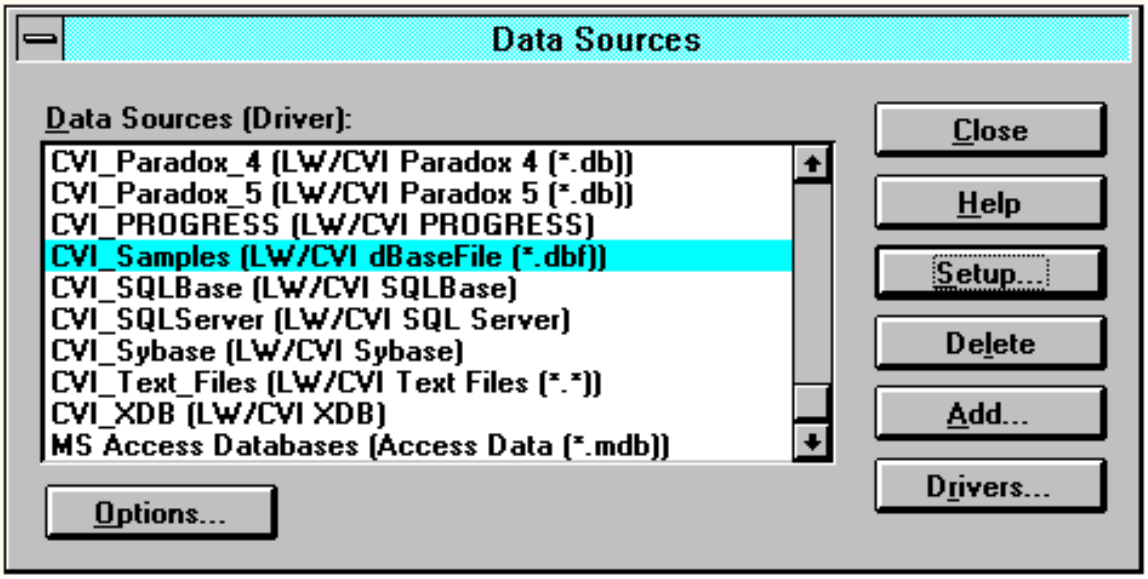

Figure 2-1. Data Sources Dialog Box

The Data Sources dialog box lists all the registered ODBC data sources. The list may include non-LabWindows/CVI SQL Toolkit ODBC Drivers, such as the Microsoft Access ODBC Driver. You can use the **Add** or **Setup** buttons to display a driver-specific dialog box that enables you to configure a new or an existing data source. The system then saves the configuration for the data source in the `ODBC.INI` file.

Figure 2-2 displays the ODBC dBASE Driver Setup dialog box.

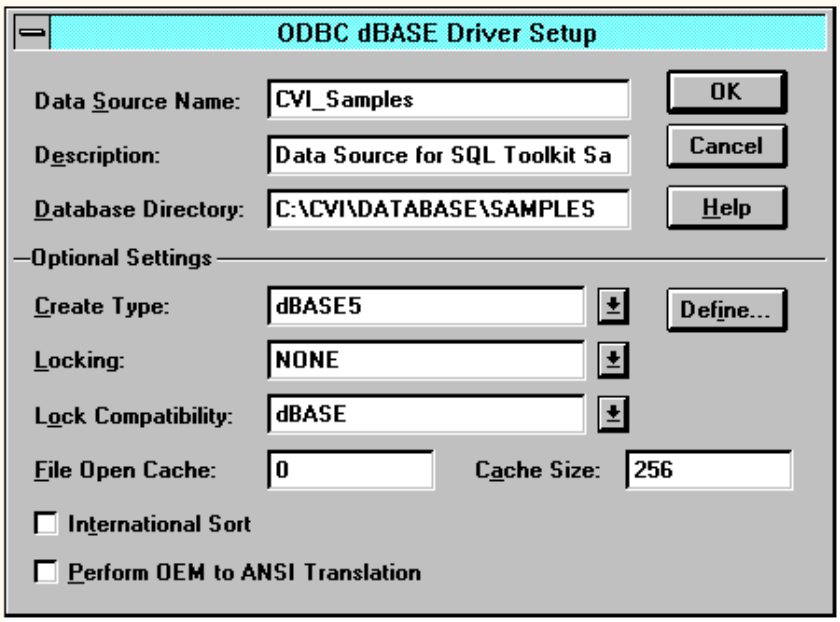

Figure 2-2. ODBC dBASE Driver Setup Dialog Box

Many of the LabWindows/CVI SQL Toolkit ODBC drivers require DLLs from other vendors. For more information on system requirements and configuring the supplied ODBC data source, please refer to the *Help on Individual Drivers* section in the online driver help file in the LabWindows/CVI SQL Toolkit program group.

Note: ***If you attempt to use an ODBC driver without the required DLLs or if you incorrectly configure an ODBC driver , the following message appears.***

```
The setup routines for the ODBC driver could not be loaded. You may
be low on memory and need to quit a few applications.
```

Third Party ODBC Database Drivers

The LabWindows/CVI SQL Toolkit supplied ODBC Database Drivers offer you a wide range of support for industry accepted database formats. Because the LabWindows/CVI SQL Toolkit complies with the ODBC standard, you can also use other ODBC compliant database drivers. Please refer to your third party vendor documentation for more information on registering your specific database drivers with the ODBC Administrator.

The Database Session

Database interactions occur within a database session. A simple session consists of the following steps.

1. Connect to the database.
2. Activate SQL statements.
3. Process SQL statements.
4. Deactivate SQL statements.
5. Disconnect from the database.

Connecting to the Database

Before you can execute SQL statements, you must establish a connection to a database. The SQL Toolkit supports multiple simultaneous connections to a single database or to multiple databases.

Activating SQL Statements

With the SQL Toolkit, you can use two methods for activating statements—automatic SQL and explicit SQL.

- Automatic SQL constructs the statement for you. Automatic SQL can only construct simple `SELECT` and `CREATE TABLE` statements.
- Explicit SQL must have the statement passed into the function. Use explicit SQL for more complex `SELECT` statements or other types of statements.

For more details on automatic SQL and explicit SQL, see Chapter 3, *Using the SQL Toolkit*.

Processing SQL Statements

In general, only SQL `SELECT` statements require further processing. `SELECT` statements are important components of the SQL Toolkit. You use `SELECT` statements for the following database operations.

- Retrieving rows from a table.
- Updating rows in a table.
- Creating new rows in a table.

To use a SELECT statement, you must bind the selected columns to variables in your program. You can then use the fetch functions to retrieve the selected rows. Each time you call a fetch function, the SQL Toolkit copies the column values into the bound variables. You also use the bound variables when updating a row or creating a new row. That is, when updating or creating a row, you copy the new values into bound variables and then call the appropriate function. For more details on variable binding, see Chapter 3, *Using the SQL Toolkit.*

You can also get information about an active SELECT statement, such as the number of columns selected, the name and data type of a given column, and the number of rows selected. You will find this information especially useful when selecting all columns (SELECT * ...) in an unfamiliar table or when creating a program, such as a database browser, that must access a variety of tables.

Deactivating SQL Statements

After you finish using a statement, you should deactivate the statement to free system resources. This deactivation is especially important when fetching in any direction so that the toolkit properly closes and deletes temporary log files.

Disconnecting from the Database

At the end of a database session, disconnect from the database to free system resources.

Note: ***To prevent problems in cases of program failure, the SQL Toolkit will deactivate any remaining SQL statements and disconnect any remaining connections when program execution ends. However, for the best use of resources, you should explicitly deactivate statements and connections after completing each session.***

Chapter 3
Using the SQL Toolkit

This chapter describes how to use the SQL Toolkit functions for common types of database operations and contains example code for performing each operation.

Function Summary

Figure 3-1 shows how the major SQL Toolkit functions relate to each other. The rest of this chapter describes the steps in this illustration.

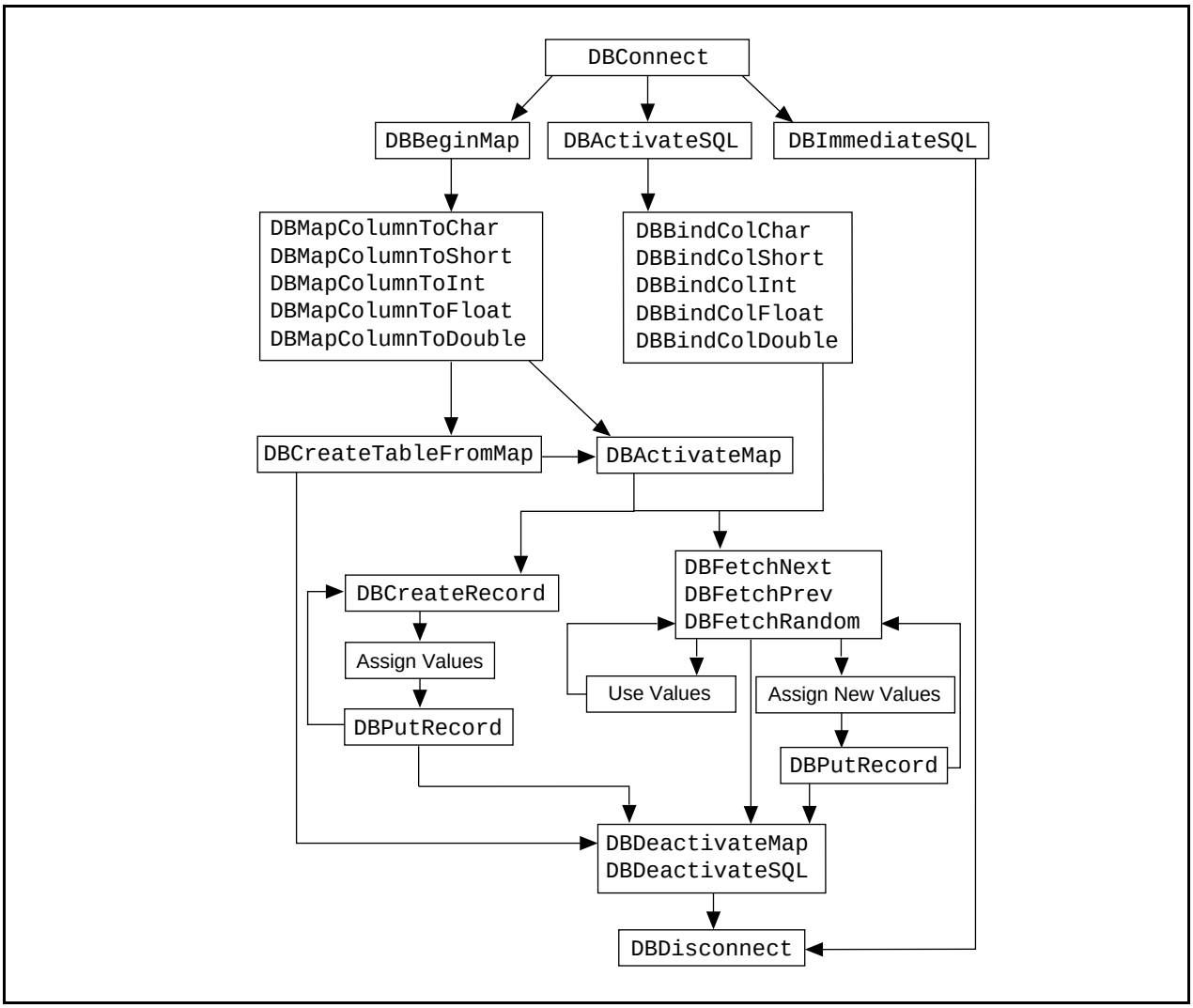

Figure 3-1. The SQL Toolkit Functions

Table 3-1 provides the data that the examples in the rest of this chapter are based on.

Table 3-1. Sample Database Table

UUT_NUM	MEAS1	MEAS2
20860B456	0.5	0.6
20860B456	1.2	
20860B123	-0.1	0.7
20860B789	0.6	0.6
20860B789	1.3	

Connecting to a Database

Use `DBConnect` to connect to a data source. The only parameter is a connection string, which must contain the data source name and any other attributes the database requires. `DBConnect` returns a handle to the database connection that other functions in the toolkit use. Call `DBDisconnect` to close the database connection, passing in the database connection handle from `DBConnect`. You can find the following example in `connect.prj`.

```
int hdbc = 0;    /* Handle to database connection */

/* Connect to CVI_Samples data source */

hdbc = DBConnect ("DSN=CVI Samples");
if (hdbc <= 0) goto Error;
...

/* Disconnect from database */

resCode = DBDisconnect (hdbc);
if (resCode != DB_SUCCESS) goto Error;
```

Using Automatic SQL (Maps)

The following example uses the SQL Toolkit mapping functions to automatically generate and execute a SQL `SELECT` statement. See `readtab.prj` for the complete program. To define a map, first call `DBBeginMap`. The only parameter consists of the connection handle from `DBConnect`. The return value acts as a handle to the map.

Next, you must map the columns of interest to program variables with the `DBMapColumnTo` functions (in this case, `DBMapColumnToChar` and `DBMapColumnToDouble`). All the `DBMapColumnTo` functions use the map handle, the column name, the address of the target variable, and the address of the status variable as parameters. `DBMapColumnToChar` has two additional parameters, the size of the buffer and a format string. If you do not need to use

formatting, use the empty string, "" as the format string. See Appendix C, *Format Strings*, for a description of format strings.

After the program maps all the columns, call DBActivateMap to construct the SQL SELECT statement, execute the statement, and bind the columns to the variables. DBActivateMap has two parameters—the connection handle and the name of the table. The return value acts as a handle to the executed SQL statement and is a parameter to other SQL Toolkit functions. When you finish using the SQL statement, call DBDeactivateMap to free system resources. The only parameter for DBDeactivateMap consists of the map handle from DBActivateMap.

```
/* begin map for constructed SQL statement */

hmap = DBBeginMap (hdbc);
if (hmap <= 0) goto Error;
...

/* specify the columns to be selected and the variables where column */
/* values will be placed. */

resCode = DBMapColumnToChar (hmap, "UUT_NUM", 11, uutNum,
   &uutNumStat, "");
if (resCode != DB_SUCCESS) goto Error;
resCode = DBMapColumnToDouble (hmap, "MEAS1", &meas1, &meas1Stat);
if (resCode != DB_SUCCESS) goto Error;
resCode = DBMapColumnToDouble (hmap, "MEAS2", &meas2, &meas2Stat);
if (resCode != DB_SUCCESS) goto Error;
...
/* Activate the map for table testres.  */
/* (Construct a SQL SELECT statement, execute the statement, bind the */
/* selected columns to the previously specified variables.) */

hstmt = DBActivateMap (hmap, "TESTRES");
if (hstmt == 0) goto Error;
...
/* Fetching or other operations */
...
resCode = DBDeactivateMap(hmap);
if (resCode != DB_SUCCESS) goto Error;
```

You can also use DBCreateTableFromMap to create a table. The toolkit will use the same parameters as DBActivateMap, the map handle, and the table name. After creating the table, you can still use the map with DBActivateMap. For example, you can use the map to create the initial records for the table. You can find the following example in new_tabl.prj.

```
/* begin map for constructed SQL statement */

hmap = DBBeginMap (hdbc);
if (hmap <= 0) goto Error;

/* specify the columns to be selected and the variables where column */
/* values will be placed.*/

resCode = DBMapColumnToChar (hmap, "UUT_NUM", 11, uutNum,
   &uutNumStat, "");
if (resCode != DB_SUCCESS) goto Error;
resCode = DBMapColumnToDouble (hmap, "MEAS1", &meas1, &meas1Stat);
```

```
if (resCode != DB_SUCCESS) goto Error;
resCode = DBMapColumnToDouble (hmap, "MEAS2", &meas2, &meas2Stat);
if (resCode != DB_SUCCESS) goto Error;
resCode = DBCreateTableFromMap (hmap, "TESTRES");
if (resCode != DB_SUCCESS) goto Error;
...
/* Optionally activate the map and create records */
...
resCode = DBDeactivateMap(hmap);
if (resCode != DB_SUCCESS) goto Error;
```

Using Explicit SQL Statements

The following example, from `readtab2.prj`, executes a SQL `SELECT` statement. Notice that the supplied `SELECT` statement contains a `WHERE` clause, which is not possible using the mapping functions. First, `DBActivateSQL` executes the SQL statement. The parameters consist of the database connection handle from `DBConnect` and the SQL statement. The return value acts as a handle to the executed SQL statement, and is a parameter to other toolkit functions. Next, the `DBBindCol` functions bind the column values to program variables.

All `DBBindCol` functions use the statement handle, column number, the address of the target variable, and the address of the status variable as parameters. `DBBindColChar` has two additional parameters, the size of the buffer and a format string. If you do not need to use formatting, use the empty string, `""` as the format string. See Appendix C, *Format Strings*, for a description of format strings. You must bind all columns in the `SELECT` statement to variables, and you must bind the columns in the order they appear in the `SELECT` statement. When finished with the SQL statement, call `DBDeactivateSQL` to free system resources. The only parameter for `DBDeactivateSQL` consists of the statement handle from `DBActivateSQL`.

```
/* Execute a SELECT statement */

hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, MEAS1, MEAS2 FROM TESTRES
   WHERE MEAS2 > 1.0");
if (hstmt == 0) goto Error;
...
/* Bind the columns in the SELECT statement to */
/* program variables */

resCode = DBBindColChar (hstmt, 1, 11, uutNum, &uutStat, "");
if (resCode != DB_SUCCESS) goto Error;
resCode = DBBindColDouble (hstmt, 2, &meas1, &meas1Stat);
if (resCode != DB_SUCCESS) goto Error;
resCode = DBBindColDouble (hstmt, 3, &meas2, &meas2Stat);
if (resCode != DB_SUCCESS) goto Error;
...
/* Fetching or other operations */
...
resCode = DBDeactivateSQL(hstmt);
if (resCode != DB_SUCCESS) goto Error
```

If the SQL statement does not require further processing (in other words, most non-`SELECT` statements), you can use `DBImmediateSQL` to activate and deactivate the statement in one step.

This example, from new_tabl.prj, executes a SQL CREATE TABLE statement. Notice that you do not need to bind variables in this example.

```
resCode = DBImmediateSQL (hdbc,"CREATE TABLE TESTRES (UUT_NUM CHAR (10),
   MEAS1 NUMERIC (10,2), MEAS2 NUMERIC(10,2))");
   if (resCode != DB_SUCCESS) goto Error;
```

Fetching Records

Fetching records uses the same functions for automatic SQL and explicit SQL. Typically, you fetch records from first to last with DBFetchNext. The parameter for DBFetchNext consists of the statement handle from DBExecuteMap or DBActivateSQL. The return value consists of a result code. A DB_EOF result code indicates that no more records can be fetched. You can use the following code fragment with either of the previous examples to fetch the selected records. You can find this example in either readtab.prj or readtab2.prj.

```
/* Fetch the values. column values are placed in the previously */
/* bound variables. */

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
   printf("Serial Number %s measurement 1: %f measurement 2:
   %f\n",uutNum, meas1, meas2);
}
if ((resCode != DB_SUCCESS) && (resCode != DB_EOF))
   goto Error;
```

You can also fetch the previous record with DBFetchPrev or a specified record with DBFetchRandom. Use the same parameters as for DBFetchNext, except you must also pass the record number to DBFetchRandom. To use DBFetchPrev or DBFetchRandom, you must enable fetching in any direction by calling DBAllowFetchAnyDirection before activating the statement or map. The following example first fetches the last record (notice the use of DBNumberOfRecords) and then fetches the remaining records in reverse order. You can find this example as part of readtab3.prj.

```
/* allow DBFetchPrev and DBFetchxRandom */

resCode = DBAllowFetchAnyDirection (hdbc, 1);
if (resCode != DB_SUCCESS) goto Error;
...
/* activate explicit or automatic SQL statement */
...
/* Fetch the last record */
numRecs = DBNumberOfRecords(hstmt);
resCode = DBFetchRandom(hstmt, numRecs);
if (resCode != DB_SUCCESS) goto Error;
printf("Serial Number %s measurement 1: %f measurement 2: %f\n", uutNum,
    meas1, meas2);
/* Fetch the records in reverse order.*/
/* Notice that as each record is fetched, the column values are placed in the
specified variables*/
while ((resCode = DBFetchPrev (hstmt)) == DB_SUCCESS) {
   printf("Serial Number %s measurement 1: %f measurement 2: %f\n",
   uutNum, meas1, meas2);
```

```
}
if ((resCode != DB_SUCCESS) && (resCode != DB_EOF))
     goto Error;
```

Inserting a Record

You can insert a record with a SQL `INSERT` statement and `DBActivateSQL` or `DBImmediateSQL`, as in the following example from `new_rec.prj`.

```
resCode = DBImmediateSQL (hdbc, "INSERT INTO TESTRES VALUES ('2860B456',
   0.4 ,0.6)");
```

You can also insert a record with `DBCreateRecord` and `DBPutRecord`. First, you activate a map or SQL statement in the same manner as for fetching records. You then call `DBCreateRecord` with the statement handle as the only parameter. Next, copy the desired values into the bound variables for the `SELECT` statement. Finally, call `DBPutRecord` to copy the new record into the database. You can find this example in `new_rec.prj.`

```
/* Activate a map or SQL statement */
...
/* Create the new record */

resCode = DBCreateRecord (hstmt);
if (resCode != DB_SUCCESS) goto Error;

/* Put values into the bound variables */

strcpy(uutNum, "2860B456");
meas1 = 0.7;
meas2 = 1.1;

/* Insert the record into the database */

resCode = DBPutRecord (hstmt);
if (resCode != DB_SUCCESS) goto Error;
```

Updating a Record

You can update a record with a SQL `UPDATE` statement and `DBActivateSQL` or `DBImmediateSQL`. The following example comes from `update.prj`.

```
hstmt = DBActivateSQL (hdbc, "UPDATE TESTRES SET MEAS2 = 500.0 WHERE
   UUT_NUM = '2860B456'");
```

You can also update a record with `DBPutRecord`. The process is similar to inserting a record and works with either automatic SQL or explicit SQL . After activating a map or SQL statement, you must then fetch the record you wish to update. Next, copy the desired values into the bound variables. Finally, call `DBPutRecord` to copy the updated record into the database.

```
/* Activate a map or SQL statement */
```

```
...
/* Fetch the record to update */
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS)
   if (strcmp(uutNum, "2860B456  ") == 0)
      break;
if (resCode == DB_EOF)
   printf("record not found\n");
if (resCode != DB_SUCCESS) goto Error;

/* Change the value of meas2 */

meas2 = -0.5;

/* copy the updated record back to the database */

resCode = DBPutRecord (hstmt);
if (resCode != DB_SUCCESS) goto Error;
```

Deleting a Record

You can delete a record with a SQL `DELETE` statement and `DBActivateSQL` or `DBImmediateSQL`. You can find the following example in `del_rec.prj`.

```
resCode = DBImmediateSQL(hdbc,"DELETE FROM TESTRES WHERE
   UUT_NUM = '2860B567'");
```

You can also delete a record with `DBDeleteRecord`. After activating a map or SQL statement, you must then fetch the record you wish to delete. Then, call `DBDeleteRecord` to delete the record from the database.

```
/* Activate a map or SQL statement */
...
/* find and delete the record */

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
   if (strcmp(uutNum, "2860B567  ") == 0)
      resCode = DBDeleteRecord(hstmt);
}
```

Deleting a Table

You can delete a table with a SQL `DROP TABLE` statement and `DBActivateSQL` or `DBImmediateSQL.`

```
   resCode = DBImmediateSQL (hdbc,“DROP TABLE TESTRES”);
```

Note: ***You can only delete a table using SQL statements.***

Information Functions

The SQL Toolkit includes several information functions. These functions fall into two categories—available data source information and SELECT statement information.

Data Source Information

The SQL Toolkit contains three functions that return information about data sources: DBSources, DBDatabases, and DBTables. These functions all execute SELECT statements and return a statement handle that you can use to fetch the information.

- DBSources returns information about the available data source names. With the single parameter, you can choose all available data sources (DB_SRC_AVAILABLE) or only currently connected sources (DB_SRC_CONNECTED). The SELECT statement that DBSources executes returns four columns: the source name, file extension (possibly NULL), connection handle, and remarks.

- DBDatabases returns information about the available databases for a connection. The only parameter consists of the connection handle. The two columns that the SELECT returns consist of the database name and remarks. If you use DBDatabases with a flat-file database, the program will not return any records.

- DBTables returns information about the available tables. Its parameters contain the connection handle, a qualifier pattern, a user name pattern, a table name pattern, and a flags parameter. You can use % or * in the patterns to match with anything. The flags parameter chooses the type of table. The SELECT statement returns the table qualifier, table user, table name, table type, and remarks.

The following example puts the table names from DBTables into a list box. You can find this example as well as examples of DBSources and DBDatabases in pick_src.c.

```
hstmt = DBTables (hdbc, "%", "%", "%", DB_TBL_TABLE);
if (hstmt <= 0) goto Error;
resCode = DBBindColChar(hstmt, 1, 127, qual, &qualStat, "");
if (resCode != DB_SUCCESS) goto Error;
resCode = DBBindColChar(hstmt, 2, 127, user, &userStat, "");
if (resCode != DB_SUCCESS) goto Error;
resCode = DBBindColChar(hstmt, 3, 127, name, &nameStat, "");
if (resCode != DB_SUCCESS) goto Error;
resCode = DBBindColInt(hstmt, 4, &type, &typeStat);
if (resCode != DB_SUCCESS) goto Error;
resCode = DBBindColChar(hstmt, 5, 255, rem, &remStat, "");
if (resCode != DB_SUCCESS) goto Error;
while ((resCode = DBFetchNext(hstmt)) != DB_EOF) {
   if (resCode != DB_SUCCESS) {ShowError(); goto Error;}
   if (nameStat != DB_NULL_DATA)
      InsertListItem (pan, SELTABLE_TABLES, 0, name,
                           name);
}
```

SELECT Statement Information

The SQL Toolkit contains several functions that return information about SELECT statements. You will find these functions useful when you access tables without prior knowledge of the table structure. Use the following functions to return SELECT statement information.

- DBNumberOfRecords
- DBNumberOfColumns
- DBColumnName
- DBColumnType
- DBNumberOfModifiedRecords.

The first parameter for all these functions consists of the statement handle. DBColumnName and DBColumnType also have a second parameter for the column number. DBNumberOfRecords, DBNumberOfColumns, and DBNumberOfModifiedRecords return the number of items in a table.

DBColumnName returns a pointer to the column name string. Because the toolkit will reuse the buffer containing the column name, you must copy the string before you call another toolkit function. DBColumnType returns the data type of the column. You can find the following example in sel_info.prj.

```
/* We will be calling DBNumberOfRecords, so fetching in any direction */
/* must be enabled.  */

resCode = DBAllowFetchAnyDirection (hdbc, 1);
...

/* Execute a SELECT statement */
hstmt = DBActivateSQL(hdbc, selectStmt);
...

/*Get information about the columns and rows in the SELECT*/

numCols = DBNumberOfColumns (hstmt);
numRecs = DBNumberOfRecords (hstmt);
printf("Executed \"%s\"\n",selectStmt);
printf("%d rows and %d columns selected\n",numRecs, numCols);
for (i = 1; i <= numCols; i++) {
   columnName = DBColName(hstmt, i);
   columnType = DBColType(hstmt, i);
   printf("column %d: name %s type number %d\n",i,colName, colType);
}
```

Transactions

You can use the SQL Toolkit to group database changes into transactions. A transaction consists of a set of database operations that you can either commit (save) or roll back (discard). The toolkit uses the following transaction functions: `DBBeginTran`, `DBCommit`, and `DBRollback`. These functions all have one parameter, consisting of the database connection handle from `DBConnect`. You begin a transaction by calling `DBBeginTran`. After you have made changes, you can call `DBCommit` to make the changes permanent or `DBRollback` to discard the changes. Each connection can have one active transaction. This example, from `transact.prj`, starts a transaction, updates a record, and then prompts the user to either commit or rollback the transaction.

```
/* Begin transaction */
resCode = DBBeginTran(hdbc);
if (resCode != DB_SUCCESS) goto Error;
...
/* Execute SQL Statement */
hstmt = DBImmediateSQL (hdbc, "UPDATE TESTRES SET MEAS2 = 0.5 WHERE
   UUT_NUM = '2860B456'");
if (hstmt == 0) goto Error;
...
/* Other operations in the transaction */
...
/* Ask if user wants to commit the transaction */

response = ConfirmPopup ("Transaction Example", "Commit the
   transaction?");
if (response == 1)
   /* make the changes permanent */
   DBCommit(hdbc);
else
   /* discard the changes */
   DBRollback(hdbc);
```

Error Checking

The SQL Toolkit functions return one of three types of values: result codes, handles, and data. You can compare a result code with `DB_SUCCESS` to determine if an error occurred. Handles refer to such items as database connections or activated SQL statements. If a function returns a handle, a value of zero indicates an error. You can then call `DBError` to determine the error number. You can also call `DBErrorMessage` to get the text of the error message. For functions that return data, such as `DBColumnName`, you should call `DBError` to determine if an error occurred.

```
hdbc = DBConnect (hdbc);
if (hdbc == 0) {
   errorCode = DBError();
   errorMsg = DBErrorMsg();
   printf("Error number %d\n%s\n",errorCode, errorMsg);
}
```

Creating A Standalone Executable File

When executing a LabWindows/CVI standalone executable file that contains functions from the SQL Toolkit Instrument Driver, you must copy the instrument driver's `cvi_db.pth` file into the same directory as your executable file. This allows the executable file to resolve the DLL filename that the SQL Toolkit functions need to implement the database functions. You will also find a copy of the `cvi_db.pth` file in the same directory as the SQL Toolkit Instrument Driver.

Chapter 4
SQL Toolkit Function Descriptions

This chapter describes the functions in the LabWindows/CVI SQL Toolkit. The *SQL Toolkit Overview* section contains general information about the SQL Toolkit functions. The *SQL Toolkit Function Reference* section contains an alphabetical list of function descriptions.

SQL Toolkit Overview

This section contains general information about the SQL Toolkit functions.

SQL Toolkit Function Panels

The SQL Toolkit function panels are grouped in a tree structure according to the types of operations they perform. Table 4-1 shows the SQL Toolkit function tree.

The bold headings in the tree indicate the names of function classes and subclasses. Function classes and subclasses contain groups of related function panels. The headings in plain text indicate the names of individual function panels. Each function panel generates one function call. You will find the names of the corresponding function calls, in bold italics, to the right of the function panel names.

Table 4-1. The SQL Toolkit Function Tree

SQL Toolkit	
Connection	
Open Connection	***DBConnect***
Close Connection	***DBDisconnect***
Set Database	***DBSetDatabase***
Automatic SQL (maps)	
Begin Map	***DBBeginMap***
Map Column to String	***DBMapColumnToChar***
Map Column to Short Integer	***DBMapColumnToShort***
Map Column to Integer	***DBMapColumnToInt***
Map Column to Float	***DBMapColumnToFloat***
Map Column to Double	***DBMapColumnToDouble***
Create Table From Map	***DBCreateTableFromMap***
Activate Map	***DBActivateMap***
Deactivate Map	***DBDeactivateMap***

(continues)

Table 4-1. The SQL Toolkit Function Tree (Continued)

Explicit SQL	
Immediate SQL Statement	***DBImmediateSQL***
Activate SQL Statement	***DBActivateSQL***
Bind Column to String	***DBBindColChar***
Bind Column to Short Integer	***DBBindColShort***
Bind Column to Integer	***DBBindColInt***
Bind Column to Float	***DBBindColFloat***
Bind Column to Double	***DBBindColDouble***
Deactivate SQL Statement	***DBDeactivateSQL***
Fetch Records	
Fetch Next Record	***DBFetchNext***
Fetch Previous Record	***DBFetchPrev***
Fetch Random Record	***DBFetchRandom***
Allow Previous or Random Fetch	***DBAllowFetchAnyDirection***
Insert/Update/Delete Records	
Create New Record	***DBCreateRecord***
Put Record	***DBPutRecord***
Delete Record	***DBDeleteRecord***
Information Functions	
Data Source Information	
Available Sources	***DBSources***
Available Databases	***DBDatabases***
Available Tables	***DBTables***
Select Information	
Number of Records	***DBNumberOfRecords***
Number of Columns	***DBNumberOfColumns***
Column Name	***DBColumnName***
Column Width	***DBColumnWidth***
Column Type	***DBColumnType***
Number of Modified Records	***DBNumberOfModifiedRecords***
Transactions	
Begin Transaction	***DBBeginTran***
Commit Transaction	***DBCommit***
Rollback Transaction	***DBRollback***
Errors	
Error Code	***DBError***
Warning Code	***DBWarning***
Native Error Code	***DBNativeError***
Error/Warning Text	***DBErrorMessage***

The following list describes the function classes in the tree.

- The **Connection** functions control connecting to and disconnecting from a database.
- The **Automatic SQL** (Maps) functions define, construct, and execute SQL statements automatically.
- The **Explicit SQL** functions execute SQL statements provided by the programmer.
- The **Fetch Records** functions retrieve `SELECT` statement results from a database table.
- The **Insert/ Update/Delete Records** functions allow inserting new rows, updating existing rows, and deleting rows.
- The **Information** functions provide information about tables, databases, and active SQL statements.
- The **Transactions** functions allow database operations to be grouped into transactions.
- The **Errors** functions support error checking and reporting.

Include Files

The SQL Toolkit contains an include file, named `cvi_db.h`, which consists of function declarations and defined constants for all the toolkit routines. You must include this file in all code modules that reference the SQL Toolkit.

Reporting Errors

Most of the functions in the SQL Toolkit return an integer code containing the result of the call. A negative return code indicates that an error occurred. Otherwise, the function completed successfully. The remaining functions return a handle or other values. A handle value of zero indicates that an error occurred. You can call `DBError` to get the error code for the most recent SQL Toolkit function.

SQL Toolkit Function Reference

This section describes each function in the SQL Toolkit. The functions are arranged alphabetically.

DBActivateMap

`int` **statementHandle** = **DBActivateMap** (`int` **mapHandle**, `char *`**tableName**);

Purpose

Activates a map. This process includes constructing a SQL `SELECT` statement based on the map and table name, executing the statement, binding program variables to the resulting columns, and freeing resources used by the map.

Parameters

Input	**mapHandle**	integer	The handle to the map returned by `DBBeginMap`.
	tableName	string	Name of the database table the map will be applied to.

Return Value

statementHandle	integer	Returned handle to the statement execution. This value identifies the statement and is a parameter to other functions. If 0, the statement could not be executed.

DBActivateSQL

```
int statementHandle = DBActivateSQL (int connectionHandle,
                                     char *SQLStatement);
```

Purpose

Activates a SQL statement.

Parameters

Input	**connectionHandle**	integer	The handle to the database connection returned by `DBConnect`.
	SQLStatement	string	The SQL statement to activate.

Return Value

statementHandle	integer	Returned handle to the statement execution. This value identifies the statement and is a parameter to other functions. If 0, the statement could not be executed.

DBAllowFetchAnyDirection

`int` **resCode** =**DBAllowFetchAnyDirection** (`int` **connectionHandle**, `int` **enable**);

Purpose

Enables or disables fetching `SELECT` statement results in either direction for a database connection. If you use this function, you must call it before using `DBActivateSQL` or `DBActivateMap`..

Parameters

Input	**connectionHandle**	integer	The handle to the database connection returned by `DBConnect`.
	enable	integer	Using 1 enables fetching in any direction, using 0 disables fetching in any direction.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBBeginMap

`int` **mapHandle** = **DBBeginMap** (`int` **connectionHandle**);

Purpose

Begins a set of column-to-variable mappings. The map describes which columns will be selected and the variables that will receive column values when the program fetches a record. You can also use the map to define a new table for creation with `DBCreateTableFromMap`.

Parameters

Input	**connectionHandle**	integer	The handle to the database connection returned by `DBConnect`.

Return Value

mapHandle	integer	Handle to the new map. The functions `DBActivateMap` and `DBCreateTableFromMap` use this value.

DBBeginTran

`int` **resCode** = **DBBeginTran** (`int` **connectionHandle**);

Purpose

Begins a transaction on a database connection. After a transaction begins, the `SQL INSERT, UPDATE`, and `DELETE` statements that you execute using `DBActivateSQL`, `DBActivateMap`, or `DBImmediateSQL` cannot be saved to the database until you call `DBCommit`.

`DBCommit` saves the changes that have been made since the `DBBeginTran` call and frees all database locks. Alternately, `DBRollback` discards the changes that you have made since calling `DBBeginTran` and frees all database locks.

If you execute an `INSERT`, `UPDATE`, or `DELETE` statement without first calling `DBBeginTran`, the toolkit automatically saves the database changes and frees all database locks.

You cannot have more than one simultaneous transaction active on a database connection. After you call `DBBeginTran`, you must call either `DBCommit` or `DBRollback` before you call `DBBeginTran` again on the same database connection.

After you call `DBBeginTran`, you must call either `DBCommit` or `DBRollback` before you call `DBDisconnect`. Calling `DBDisconnect` with an active transaction results in an error.

Parameters

Input	**connectionHandle**	integer	The handle to the database connection returned by `DBConnect`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBBindColChar

```
int resCode = DBBindColChar(int statementHandle, int columnNumber,
                            int maxLen, char locationForValue[],
                            int *locationForStatus, char formatString[]);
```

Purpose

Specifies the value and status variables in your program that will receive a column's value and length each time the program fetches a record.

You must bind all columns in the `SELECT` statement. You must bind the columns in the order in which they occur in the statement.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL`.
	columnNumber	integer	The column number containing specified variables. The first column number is 1.
	maxLen	integer	Size of the value variable in bytes.
Output	**locationForValue**	string	Pointer to the variable that will to receive the null-terminated character string value for the column when the program fetches a record.
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or `NULL` data `DB_TRUNCATION (-1)` or `DB_NULL_DATA (-2)`.
	formatString	string	Used to control formatting for dates and numbers.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBBindColDouble

```
int resCode = DBBindColDouble(int statementHandle, int columnNumber,
                              double *locationForValue, int *locationForStatus);
```

Purpose

Specifies the value and status variables in your program that will receive the column's value and length each time the program fetches a record.

You must bind all columns in the SELECT statement. You must bind the columns in the order in which they occur in the statement.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by DBActivateSQL.
	columnNumber	integer	The column number containing specified variables. The first column number is 1.
Output	**locationForValue**	double	Pointer to the variable that will receive the double value for the column when the program fetches a record.
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or NULL data DB_TRUNCATION (-1) or DB_NULL_DATA (-2).

Return Value

resCode	integer	Result code. See DBError for a list of result codes.

DBBindColFloat

`int` **resCode** = **DBBindColFloat**(`int` **statementHandle**, `int` **columnNumber**,
`float` ***locationForValue**, `int` ***locationForStatus**);

Purpose

Specifies the value and status variables in your program that will receive the column's value and length each time the program fetches a record. When fetched, the program converts data to a single-precision, floating point value.

You must bind all columns in the statement. You must bind the columns in the order in which they occur in the statement.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL`.
	columnNumber	integer	The column number containing specified variables. The first column number is 1.
Output	**locationForValue**	float	Pointer to the variable that will receive the float value for the column when the program fetches a record.
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or `NULL` data `DB_TRUNCATION (-1)` or `DB_NULL_DATA (-2)`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBBindColInt

```
int resCode = DBBindColInt(int statementHandle, int columnNumber,
                           int *locationForValue, int *locationForStatus);
```

Purpose

Specifies the value and status variables in your program that will receive the column's value and length each time the program fetches a record. Converts data to a 4-byte integer (a `long int` or `int` in LabWindows/CVI).

You must bind all columns in the statement. You must bind the columns in the order in which they occur in the statement.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL`.
	columnNumber	integer	The column number containing specified variables. The first column number is 1.
Output	**locationForValue**	integer	Pointer to the variable that will receive the `int` value for the column when the program fetches a record.
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or `NULL` data `DB_TRUNCATION (-1)` or `DB_NULL_DATA (-2)`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBBindColShort

```
int resCode = DBBindColShort(int statementHandle, int columnNumber,
                             short *locationForValue, int *locationForStatus);
```

Purpose

Specifies the value and status variables in your program that will receive the column's value and length each time the program fetches a record.

You must bind all columns in the statement. You must bind the columns in the order in which they occur in the statement.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL`.
	columnNumber	integer	The column number containing specified variables. The first column number is 1.
Output	**locationForValue**	short integer	Pointer to the variable that will receive a `short int` value for the column when the program fetches a record..
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or `NULL` data `DB_TRUNCATION (-1)` or `DB_NULL_DATA (-2)`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBColumnName

`char *`**columnName** = **DBColumnName**(`int` **statementHandle**, `int` **columnNumber**);

Purpose

Returns the name of a column.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL` or `DBActivateMap`.
	columnNumber	integer	The column number for which the name will be returned. The first column number is 1.

Return Value

columnName	string	Pointer to the returned column name. The SQL Toolkit maintains a buffer that stores the string. You must copy the string out of this buffer before you call another toolkit function because the next function may use the same buffer. The column name is "" if the column is an expression in the SQL statement.

DBColumnType

`int` **dataType** = **DBColumnType** (`int` **statementHandle**, `int` **columnNumber**);

Purpose

Returns the SQL Toolkit data type for a column in the SQL `SELECT` statement. Notice that this data type may not be the same as the type in the underlying database.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL` or `DBActivateMap`.
	columnNumber	integer	The column number for which the data type will be returned. The first column number is 1.

Return Value

dataType	integer	The returned data type. Possible values are: `DB_CHAR 1`—Fixed length character string; `DB_VARCHAR 2`—Variable length character string; `DB_DECIMAL 3`—Decimal Number (BCD); `DB_INTEGER 4`—Long Integer (4-byte); `DB_SMALLINT 5`—Integer (2-byte); `DB_FLOAT 6`—Floating point number (4-byte); `DB_DOUBLEPRECISION 7`—Double precision floating point number (8-byte); `DB_DATETIME 8`—Date-time (26-byte `char` string).

DBColumnWidth

`int` **colWidth** = **DBColumnWidth**(`int` **statementHandle**, `int` **columnNumber**);

Purpose

Returns the width of a column. The width consists of the size, in bytes, of the longest value that can be stored in the column.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL` or `DBActivateMap`.
	columnNumber	integer	The column number for which the width will be returned. The first column number is 1.

Return Value

colWidth	integer	The size of the longest value, which can be stored in the column in bytes.

DBCommit

`int` **resCode**= **DBCommit**(`int` **connectionHandle**);

Purpose

Saves all changes that you have made using the SQL statements `INSERT`, `UPDATE`, or `DELETE` since calling `DBBeginTran`. You must call `DBBeginTran` to begin a transaction before you can call `DBCommit` to save all changes.

Parameters

Input	**connectionHandle**	string	The handle to the database connection returned by `DBConnect`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBConnect

`int` **connectionHandle**= **DBConnect** (`char *` **connectionString**);

Purpose

Opens a connection to a database system so you can execute SQL statements.

Parameters

Input	**connectionString**	string	Identifies the database system and any additional login information. The connection string has the form:
			`DSN=<data source name>`
			`[;<attribute>=<value>`
			`[;<attribute>=<value>]`
			`...]`
			The following list consists of the most commonly used attributes. (For additional attributes, see the documentation for the particular driver).
			`DSN`—Name of the data source defined in the `ODBC.INI` file.
			`DLG`—When enabled `(DLG=1)`, displays a dialog box that allows user input of connection string information.
			`UID`—The user ID or name.
			`PWD`—The password.
			`MODIFYSQL`—Set to 1, support
			`ODBC` compliant SQL. Set to 0, support native SQL of the underlying database.

Return Value

connectionHandle	integer	The returned handle to the database connection. This value identifies the connection and acts as a parameter to other functions. If the handle is 0, the connection could not be opened.

DBCreateRecord

`int` **resCode**= **DBCreateRecord**(`int` **statementHandle**);

Purpose

Creates the buffer that will be used for a new record. All column values are initially set to `NULL`. You can put values into the buffer by copying the values into bound variables. You can then insert the record in the database by calling `DBPutRecord.`

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL` or `DBActivateMap.`

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBCreateTableFromMap

`int` **resCode** = **DBCreateTableFromMap**(`int` **mapHandle**, `char` ***tableName**);

Purpose

Creates a database table based on a map. This process involves constructing a SQL `CREATE TABLE` statement from the map and table name and then executing the statement.

Parameters

Input	**mapHandle**	integer	The handle to the map returned by `DBBeginMap`.
	tableName	string	Name of the database table to which the map will be applied.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBDatabases

`int` **statementHandle**= **DBDatabases** (`int` **connectionHandle**);

Purpose

Creates and activates a `SELECT` statement that returns information about the available databases on a connection. You can then use the `DBBindCol` and `DBFetch` functions to retrieve the information. Table 4-2 shows the two columns that each record contains.

Table 4-2. Columns Contained in Each Record

Column	Type	Description
Database	Char (128)	Database name.
Remarks	Char (256)	Remarks (may be `NULL`).

Note: ***If you use*** `DBDatabases` ***with a flat-file database, it will not select any records.***

Parameters

Input	**connectionHandle**	integer	The handle to the database connection returned by `DBConnect`.

Return Value

statementHandle	integer	Handle to the SQL statement execution. If 0, the statement could not be executed.

DBDeactivateMap

`int` **resCode** = **DBDeactivateMap**(`int` **mapHandle**);

Purpose

Ends activation of a map. You should use `DBDeactivateMap` to free system resources when you finish using the map.

Parameters

Input	**mapHandle**	integer	The handle to the map returned by `DBBeginMap`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBDeactivateSQL

`int` **resCode** = **DBDeactivateSQL**(`int` **statementHandle**);

Purpose

Deactivates a SQL statement. You should use `DBDeactivateSQL` to free system resources when you finish using the statement.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBDeleteRecord

`int` **resCode** = **DBDeleteRecord**(`int` **statementHandle**);

Purpose

Deletes the current record.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL` or `DBActivateMap`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

Note: ***After deleting a record, the current record will be positioned between the previous record and the next record in the buffer. You must call*** `DBFetchNext` ***after deleting a record to position on the next record.***

DBDisconnect

`int` **resCode** = **DBDeactivateMap**(`int` **connectionHandle**);

Purpose

Closes a connection to a database system. You should close all connections before your program terminates to free system resources used by the connection.

`DBDisconnect` will end execution of any active SQL statements on the connection.

Parameters

Input	**connectionHandle**	integer	Handle to the database connection returned by `DBConnect`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBError

`int` **errorCode** = **DBError**(`void`);

Purpose

Returns the result code of the last SQL Toolkit function you called.

You should call `DBError` immediately after calling any other SQL Toolkit function that does not return a result code (for example, `DBColumnName`).

Parameters

None

Return Value

errorCode	integer	Result code of the last SQL Toolkit function called. Possible values are: `DB_HSTMT_BUSY (-7)` Returned when a thread of execution attempts to use a statement handle that is currently in use by another thread of execution. `DB_LOCK_NO_REC (-6)` Attempted a lock, but either no record was selected by the primary key, the record was deleted by another user, or another user changed the value of a key field. `DB_EOF(-5)` `EOF` returned by `DBFetchNext`, `DBFetchPrev`, or `DBFetchRandom` when there is no record to return. `DB_USER_CANCELED (-4)` User canceled out of the box. `DB_OUT_OF_MEMORY (-3)` Not enough memory in Windows. This is usually fatal. `DB_SUCCESS (0)` Success. `DB_SUCCESS_WITH_INFO (1)` Success with warning.

(continues)

Return Value (Continued)

errorCode	integer	`DB_NO_DATA_WITH_INFO (2) EOF` with additional information (usually occurs if you hit `ESC` during a fetch). `DB_DBSYS_ERROR (4)` Database system error. `DB_LIBSYS_ERROR (5)` Returned when the system cannot locate the DataDirect Dynamic Link Library. <other errors> (1000+) Call `DBErrorMsg` to get the text of the error message.

DBErrorMessage

`char *` **errorMessage** = **DBError**(`void`);

Purpose

Returns the text associated with the error or warning generated by the last SQL Toolkit function you called.

Parameters

None

Return Value

errorMessage	string	Pointer to the returned error message. The SQL Toolkit maintains a buffer that stores the string. You must copy the string out of this buffer before you call another toolkit function becuase the next function may use the same buffer.

DBFetchNext

`int` **resCode** = **DBFetchNext**(`int` **statementHandle**);

Purpose

Retrieves the next record from the database. The program places the column values in the variables previously specified by the variable binding or mapping functions. You can use `DBFetchNext` with either automatic SQL or explicit SQL.

When `DBFetchNext` attempts to fetch a record beyond the last record returned by the `SELECT` statement, it returns a result of `DB_EOF (-5)`.

Parameters

Input	**statementHandle**	integer	Handle to the database connection returned by `DBConnect`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBFetchPrev

`int` **resCode** = **DBFetchPrev**(`int` **statementHandle**);

Purpose

Retrieves the previous record from the database. The program places the column values in the variables previously specified by the variable binding or mapping functions. You can use `DBFetchPrev` with either automatic SQL or explicit SQL.

You can use this function only if you have previously called `DBAllowFetchAnyDirection` to allow fetches in both directions.

When `DBFetchPrev` attempts to fetch a record before the first record returned by the `SELECT` statement, it returns a result of `DB_EOF (-5)`.

Parameters

Input	**statementHandle**	integer	Handle to the database connection returned by `DBConnect`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBFetchRandom

`int` **resCode** = **DBFetchRandom**(`int` **statementHandle**, `int` **recordNumber**);

Purpose

Retrieves the designated record from the database. The program places the column values in the variables previously specified by the variable binding or mapping functions. You can use `DBFetchRandom` with either automatic SQL or explicit SQL.

You can use this function only if you have previously called `DBAllowFetchAnyDirection` to allow fetches in both directions.

When `DBFetchRandom` attempts to fetch a record not contained in the result set returned by the `SELECT` statement, it returns a result of `DB_EOF (-5)`.

Parameters

Input	**statementHandle**	integer	Handle to the database connection returned by `DBConnect`.
	recordNumber	integer	The record number to be read. The first record is 1.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBImmediateSQL

`int` **resCode** = **DBImmediateSQL**(`int` **connectionHandle**, `char *`**SQLStatement**);

Purpose

Executes a SQL statement immediately. Equivalent to `DBActivateSQL`, followed by `DBDeactivateSQL`. This function is useful for any SQL statement that does not require further processing such as `CREATE TABLE, INSERT`, and `UPDATE`. Because this function also ends statement execution, it is not useful for `SELECT` statements.

Parameters

Input	**connectionHandle**	integer	The handle to the database connection returned by `DBConnect`.
	SQLStatement	string	The SQL statement to be executed.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBMapColumnToChar

```
int resCode = DBMapColumnToChar(int mapHandle, char * columnName,
                                int maxLen, char locationForValue[],
                                int *locationForStatus, char formatString[]);
```

Purpose

Specifies a column to be selected and the value and status variables that will receive the column's value and status each time the program fetches a record.

Parameters

Input	**mapHandle**	integer	Handle to the map returned by `DBBeginMap`.
	columnName	string	The name of the column containing specified variables.
	maxLen	integer	Size of the value variable in bytes.
Output	**locationForValue**	string	Pointer to the variable that will receive the null-terminated character string value for the column when the program fetches a record.
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or `NULL` data `DB_TRUNCATION (-1)` or `DB_NULL_DATA (-2)`.
	formatString	string	Used to control formatting for dates and numbers.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBMapColumnToDouble

```
int resCode = DBMapColumnToChar(int mapHandle, char * columnName,
                                double *locationForValue,
                                int *locationForStatus);
```

Purpose

Specifies a column to be selected and the value and status variables that will receive the column's value and status each time the program fetches a record.

Parameters

Input	**mapHandle**	integer	Handle to the map returned by `DBBeginMap`.
	columnName	string	The name of the column containing specified variables.
Output	**locationForValue**	double precision	Pointer to the variable that will receive the `double` value for the column when the program fetches a record.
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or `NULL` data `DB_TRUNCATION (-1)` or `DB_NULL_DATA (-2)`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBMapColumnToFloat

```
int resCode = DBMapColumnToFloat(int mapHandle, char * columnName,
                                 float *locationForValue,
                                 int *locationForStatus);
```

Purpose

Specifies a column to be selected and the value and status variables that will receive the column's value and status each time the program fetches a record.

Parameters

Input	**mapHandle**	integer	Handle to the map returned by `DBBeginMap`.
	columnName	string	The name of the column containing specified variables.
Output	**locationForValue**	float	Pointer to the variable that will receive the `float` value for the column when the program fetches a record.
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or `NULL` data `DB_TRUNCATION (-1)` or `DB_NULL_DATA (-2)`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBMapColumnToInt

```
int resCode = DBMapColumnToInt(int mapHandle, char * columnName,
                               int *locationForValue,
                               int *locationForStatus);
```

Purpose

Specifies a column to be selected and the value and status variables that will receive the column's value and status each time the program fetches a record.

Parameters

Input	**mapHandle**	integer	Handle to the map returned by `DBBeginMap`.
	columnName	string	The name of the containing specified variables.
Output	**locationForValue**	integer	Pointer to the variable that will receive the `int` value for the column when the program fetches a record.
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or `NULL` data `DB_TRUNCATION (-1)` or `DB_NULL_DATA (-2)`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBMapColumnToShort

```
int resCode = DBMapColumnToShort(int mapHandle, char * columnName,
                                 short *locationForValue,
                                 int *locationForStatus);
```

Purpose

Specifies a column to be selected and the value and status variables that will receive the column's value and status each time the program fetches a record.

Parameters

Input	**mapHandle**	integer	Handle to the map returned by `DBBeginMap`.
	columnName	string	The name of the column containing specified variables.
Output	**locationForValue**	short	Pointer to the variable that will receive the `short int` value for the column when the program fetches a record.
	locationForStatus	integer	Pointer to the variable that will receive the column's status when the program fetches a record. After fetching a record, you can use this variable to determine whether the fetch retrieved truncated or `NULL` data `DB_TRUNCATION (-1)` or `DB_NULL_DATA (-2)`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBNativeError

`int` **nativeErrCode** = **DBNativeError**(`void`);

Purpose

Returns the result code from the underlying database system for the last SQL Toolkit function you called.

You can call `DBNativeError` when `DBError` returns a value of `DB_DBSYS_ERROR (4)` to determine the native error code.

Parameters

None

Return Value

nativeErrCode	integer	Native error code in the underlying database system. If 0, no database system error was reported. If `DBError` returned `DB_DBSYS_ERROR (4)`, `DBNativeError` may return 0. In this case, the underlying database system does not have a separate error code.

DBNumberOfColumns

`int` **numCols** = **DBNumberOfColumns**(`int` **statementHandle**);

Purpose

Returns the number of columns selected by a SQL `SELECT` statement.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL` or `DBActivateMap`.

Return Value

numCols	integer	Number of columns selected.

DBNumberOfModifiedRecords

`int` **numModRecs** = **DBNumberOfModifiedRecords**(`int` **statementHandle**);

Purpose

Returns the number of records updated by the last function that changed the database.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL` or `DBActivateMap`.

Return Value

numModRecs	integer	Number of modified records. Returns 0 if the statement is not a `SELECT` statement.

DBNumberOfRecords

`int` **numRecs** = **DBNumberOfRecords**(`int` **statementHandle**);

Purpose

Returns the number of records chosen by the `SELECT` statement.

To determine the number of records selected, the SQL Toolkit fetches all rows from the result set. If you have not enabled backward fetching, calling `DBNumberOfRecords` causes an error to be returned. If you have selected a large number of records, this function may work slowly and create large temporary log files.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL` or `DBActivateMap`.

Return Value

numRecs	integer	Number of selected records.

DBPutRecord

`int` **resCode** = **DBPutRecord**(`int` **statementHandle**);

Purpose

Places the current record in the database. You can use `DBPutRecord` with new records created by `DBCreateRecord`, or with existing records fetched from a `SELECT` statement.

You can call `DBNumberOfModifiedRecords` to determine the number of records affected by a call to `DBPutRecord`.

Parameters

Input	**statementHandle**	integer	Handle to the SQL statement returned by `DBActivateSQL` or `DBActivateMap`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBRollback

`int` **resCode**= **DBRollback**(`int` **connectionHandle**);

Purpose

Discards all changes that you have made using the SQL statements `INSERT`, `UPDATE`, or `DELETE` since you called `DBBeginTran`. You must call `DBBeginTran` to begin a transaction before you can call `DBRollback` to undo all changes.

Note: ***The discarded changes include any saved changes on records other than the current record, any records created by calling*** `DBCreateRecord`***, and any new values placed in the current record by calls to*** `DBPutRecord`.

After calling `DBRollback`, the SQL Toolkit will be positioned between what was the last current record in the transaction and the next record in the statement handle. Before you perform any operations against the records, call one of the `DBFetch` functions to position on a valid record.

Parameters

Input	**connectionHandle**	integer	The handle to the database connection returned by `DBConnect`.

Return Value

resCode	integer	Result code. See `DBError` for a list of result codes.

DBSetDatabase

`int` **statementHandle** = **DBSetDatabase** (int **connectionHandle**, char **databaseName**[]);

Purpose

Sets the default database in systems that allow tables to be stored in separate databases. A limited number of database systems support this function.

Parameters

Input	**connectionHandle**	integer	Handle to the database connection previously returned by `DBConnect`.
	databaseName	string	Name of the new default database.

Return Value

statementHandle	integer	Result code returned by `DBDisconnect`. Result codes are the same as those returned by `DBError`.

DBSources

`int` **statementHandle** = `int` **DBSources** (`int` **option**);

Purpose

Creates and activates a `SELECT` statement that returns information about the available database sources. You can then use the `DBBindCol` and `DBFetch` functions to retrieve the information. Table 4-3 shows the columns contained in each record.

Table 4-3. Columns Contained in Each Record

Column	Type	Description
Name	Char (32)	Source name.
Extension	Char (32)	File Extension (may be `NULL`).
Connection Handle	Short integer	Connection handle of connection if connected to this source. Returns 0 if not connected.
Remarks	Char (256)	Remarks (may be `NULL`).

Parameters

Input	**option**	integer	Determines which sources will be returned. The possible values are: `DB_SRC_AVAILABLE (1)`, which returns all sources and `DB_SRC_CONNECTED (2),` which returns only currently connected sources.

Return Value

statementHandle	integer	Returned handle to the statement execution. This value identifies the statement and is a parameter to other functions. If 0, the statement could not be executed.

DBTables

`int` **StatementHandle**= **DBTables** (`int` **connectionHandle**, `char` **qualifierPattern**[], `char` **userPattern**[], `char` **tablePattern**[], `int` **flags**);

Purpose

Creates and activates a `SELECT` statement that returns information about the available tables on a connection. You can then use the `DBBindCol` and `DBFetch` functions to retrieve the information. Table 4-4 shows the columns contained in each record.

Table 4-4. Columns Contained in Each Record

Column	Type	Description
Table Qualifier	Char (128)	Qualifier for returned table.
Table User	Char (128)	User name (table-based source).
Table Name	Char (128)	Table name (table-based source).
Directory Name	Char(128)	Directory name (file-based source).
File Name	Char(128)	File name (file-based source).
Table Type	Short	Type of table: `DB_TBL_TABLE`, `DB_TBL_VIEW`, `DB_TBL_SYNONYM`, `DB_TBL_PROCEDURE`, or `DB_TBL_SYSTABLE`.
Remarks	Char (256)	Remarks (may be `NULL`).

Parameters

Input	**connectionHandle**	integer	The handle to the database connection returned by `DBConnect`.
	qualifierPattern	string	Pointer to a string containing a qualifier or path for the tables to be selected.

(continues)

Parameters (Continued)

	userPattern	string	Pointer to a string containing the pattern for selecting users. If `NULL`, the system returns tables for the current user. If the pattern is % or *, returns tables for all users. File-based databases will ignore this parameter.
	tablePattern	string	Pointer to a string containing a pattern for selecting tables or files. To select all tables, use a pattern of % or *.
	flags	integer	Specifies the type(s) of table(s) for which information will be returned. You can use any of the following values: `DB_TBL_TABLE 0x0001` Table names . `DB_TBL_VIEW 0x0002` View names. `DB_TBL_PROCEDURE 0x0004` Stored procedure names. `DB_TBL_SYSTABLE 0x0008` System table names. `DB_TBL_SYNONYM 0x0010` Synonym names. `DB_TBL_DATABASE 0x0080` Database names. Except for `DB_TBL_DATABASE`, you can combine these constants by adding them together or joining them with an “or” clause.

Return Value

statementHandle	integer	Returned handle to the statement execution. This value identifies the statement and is a parameter to other functions. If 0, the statement could not be executed.

DBWarning

`int` **warningCode** = **DBError**(void);

Purpose

Returns the warning generated by the last SQL Toolkit or DataDirect function you called. `DBWarning` is usually called after `DBError` to determine if the database system or the last function returned any warnings.

Parameters

None

Return Value

resCode	integer	Result code, either a warning code returned by the database system or one of the following. The possible variables are: `DB_LOCK_CHANGE_REC (-8)` A lock was obtained, but the record has been changed since it was originally read. This occurs only for database systems requiring a log file. `DB_LOCK_MULTI_REC (-7)` A lock was obtained, but more than one record was locked. This occurred because the primary key fields caused more than one record to be selected. `DB_NULL_DATA (-2)` The fetched value was `NULL`. `DB_TRUNCATION (-1)` A fetched value was truncated because the value size exceeded the buffer.

Appendix A
SQL Reference

This appendix briefly explains SQL commands, operators, and functions. This version of SQL is included in the ODBC standard and is applies to all ODBC-compliant databases.

SQL Commands

Table A-1 lists the SQL commands you can use with `DBActivateSQL` and `DBImmediateSQL`.

Table A-1. SQL Commands

SQL Command	Syntax	Description	Example
CREATE TABLE	CREATE TABLE table name (column def, column def,...)	Creates a new database table.	CREATE TABLE testres (uut_num char(10) NOT NULL, meas1 NUMBER (10,2) meas2 NUMBER (10,2)
DELETE	DELETE table name [WHERE where clause]	Removes rows from a database table. The "where" clause selects specific rows to delete.	DELETE testres WHERE meas1 < 0.0
DROP TABLE	Drop Table table_name	Removes a database table.	DROP TABLE testres
INSERT	INSERT table_name [options] [(col_name, col_name,...)]VALUES (expr, expr...)	Creates a new record, places data values into its columns. The VALUES clause specifies the values.	INSERT testres (uut_num, meas1, meas2) VALUES (2860C890, 0.4, 0.6)

(continues)

Table A-1. SQL Commands (Continued)

SQL Command	Syntax	Description	Example
SELECT	SELECT [DISTINCT] {* | col_expr, col_expr...} FROM {from clause} [WHERE where_clause] [GROUP BY {group clause}} [HAVING {having clause}] [UNION [ALL] (SELECT...)] ORDER BY {order_clause,...}] [FOR UPDATE OF {col_expr,...}]	Query specifies columns from tables.	SELECT uut_num, meas1 FROM testres WHERE meas1 < 0 ORDER BY uut_num DESC
UPDATE	UPDATE table_name [options] SET col_name = expr,... [WHERE where_clause]	Sets columns in existing rows to new values.	UPDATE testres SET meas2 = (meas1 + 0.1) where meas1 < 0

SQL Objects

Table A-2 lists SQL objects, which are the building blocks for SQL statements.

Table A-2. SQL Objects

Object	Description	Examples
table_name	Describes the target table name of the operation (for file-based databases, may include full path).	testres c:\cvi\database\testres.dbf
col_name	Refers to a column in a table. Some databases restrict column names.	uut_num meas1
col_expr	Specifies a single column name or a complex combination of column names, operators, and functions.	uut_num meas1 + meas2 `LOWER`(uut_num)
sort_expr	Any column expression.	
data_type	Specifies a column's data type.	`CHAR (30) NUMBER (10.5)`
constraint	Constrains the contents of a column.	`NOT NULL`
column_defn	Describes a column to create in a new table. Consists of col_name, data_type, and (optional) constraint.	uut_num `CHAR(10) NOT NULL` meas1 `NUMBER (10.5)`
char_expr	Any expression that yields a character data type.	`'PASSED'` `STR`(42.6, 10, 2)
date_expr	Any expression that yields a date data type.	`DATE()`
num_expr	Any expression that yields a number data type.	meas1 + meas2
logical_expr	Any expression that yields a logical data type.	
expr	Any expression.	

SQL Clauses

Table A-3 lists the types of clauses you can use in SQL statements.

Table A-3. SQL Clauses

Name/Syntax	Applicable Commands	Description	Examples
FROM table_name [options] [table alias]	SELECT DELETE	Specifies table name; may be a full path name for file-based databases.	SELECT * FROM testres
WHERE expr1 comparison_oper expr2 [logical_oper expr3 comparison_oper expr4]...	SELECT DELETE UPDATE	Specifies conditions that apply to each row in the table to determine an active set of rows.	SELECT * FROM testres WHERE meas1 < 0.0 and meas2 > 1.0
GROUP BY col_expr {col_expr,...}	SELECT	Specifies column(s) to apply to group active set rows.	SELECT * FROM testres GROUP BY meas1
HAVING expr1 comparison_oper expr2	SELECT used with GROUP BY	Specifies conditions to apply to group active set rows. GROUP BY must be specified first.	SELECT * FROM testres GROUP BY uut_num HAVING meas1 < 0
ORDER BY {sort_expr [DESC | ASC]}...	SELECT	Specifies row order in the active set of rows.	SELECT * FROM testres ORDER BY uut_num DESC
FOR UPDATE OF col_name [col_name...]	SELECT	Locks columns in selected rows for updates for deletion.	SELECT * FROM testres FOR UPDATE OF meas1, meas2

SQL Operators

Table A-4 lists the operators you can use in SQL statements.

Table A-4. SQL Operators

Operator Class and Operators	Description	Examples
Constants		
	Numeric constant.	1234, 1234.5678
‘ ‘ “ “	Character constant.	‘PASSED’, “CVI”
{}	Date-time constant.	{2/8/60},{16:59:59}
.T. .F.	Logical constant.	.T., .F.
Numeric		
()	Operator precedence.	(meas1 + meas2) * (meas3 - meas4)
+ -	Sign.	- meas1
* /	Multiply/divide.	meas1 * meas2, meas1 / meas2
+ -	Add/subtract.	meas1 + meas2, meas1 - meas2
** ^	Exponentiation.	meas1 ** power, meas1 ^ 2
Character		
+	Concatenate. (keep trailing blanks)	‘keep ‘ + ‘space’ (result: ‘keep space’)
-	Concatenate. (drop trailing blanks)	‘drop ‘ - ‘space’ (result:‘dropspace’)
Comparison		
=	Equal.	`WHERE` meas1 = meas2
<>	Not equal.	`WHERE` meas1 <> meas2
>=	Greater than or equal.	`WHERE` meas1 >= meas2
<=	Less than or equal.	`WHERE` meas1 <= meas2
`IN`	Contained in the set().	`WHERE` uut_num IN (‘2860A123’,’2860A1234’)
`[NOT] IN`		`WHERE` result `NOT IN (‘FAILED’, ‘RETEST’)`
`ANY, ALL`	Compare with list of rows.	`WHERE` uut_num = `ANY (SELECT...)`
`BETWEEN`		`WHERE` meas1 `BETWEEN` 0.0 `AND` 1.0
`EXISTS`		
`[NOT] LIKE`	Within value range.	`WHERE EXISTS (SELECT...)`
		`WHERE` uut_num `LIKE ‘TEK%’`

(continues)

Table A-4. SQL Operators (Continued)

Operator Class and Operators	Description	Examples
Comparison (cont.)		
[NOT] NULL	Existence of at least one row character pattern match empty.	WHERE uut_num NOT NULL
Date		
+ -	Add/subtract.	testdate + 5 {result: new date} testdate - {2/8/60} (result:number of days)
Logical		
()	Precedence.	WHERE (res1 AND res2) OR (res3 AND res4)
NOT	Negation.	WHERE NOT (uut_num IN (SELECT...))
AND	And.	WHERE meas1 < 0.0 AND meas2 > 1.0
OR	Or.	WHERE meas1 < 0.0 OR meas2 < 1.0
Set		
UNION	Set of all rows from all individual distinct queries.	SELECT ... UNION SELECT...
Other		
*	All columns.	SELECT * FROM testres
COUNT(*)	Count of all rows.	SELECT COUNT(*) FROM testres
DISTINCT	Only non-duplicate rows.	SELECT DISTINCT FROM...

SQL Functions

Table A-5 lists the functions you can use in SQL statements.

Table A-5. SQL Functions

Function	Description
ROUND(num_expr1, num_expr2)	num_expr1 rounded to num_expr2 decimal places.
CHR(num_expr)	Character having ASCII value num_expr.
LOWER(char_expr)	Change all characters in char_expr to lower case.
LTRIM(char_expr)	Strip leading spaces from char_expr.
LEFT(char_expr)	Leftmost character of char_expr.
RIGHT(char_expr)	Rightmost character of char_expr.
SPACE(num_expr)	Construct a string with num_expr blanks.
IFF(logical_expr, True_Value, False_Value)	Return True_Value if logical_expr is true otherwise return False_Value.
STR(num_expr, width [prec])	Converts num_expr to string of width characters with optional prec fractional digits.
STRVAL(expr)	Converts any expr to a character string.
TIME()	Returns time of day as a character string.
LEN(char_expr)	Number of characters in char_expr.
AVG(column_name)	Average of all non-null values in column_name.
(must be numeric column)	
COUNT(*)	Number of rows in table.
MAX(col_expr)	Maximum value of col_expr.
MAX(num_expr1, num_expr2)	Maximum of num_expr1 and num_expr2.
MIN(num_expr1, num_expr2)	Minimum of num_expr1 and num_expr2.
SUM(col_expr)	Sum of values in col_expr.

(continues)

Table A-5. SQL Functions (Continued)

Function	Description
DTOC(date_expr, fmt_value[, separator_char])	Convert date_expr to character string using fmt and optional separator_char. fmt_values are: 0: MM/DD/YY 1: DD/YY/MM 2: YY/MM/DD 10: MM/DD/YYYY 11: DD/MM/YYYY 12: YYYY/MM/DD
USERNAME()	Returns name of current user (not supported by all databases).
MOD(num_expr1, num_expr2)	Remainder of num_expr1 divided by num_expr2.
MONTH(date_expr)	Returns month from date_expr as a number.
DAY(date_expr)	Returns day from date_expr as a number.
YEAR(date_expr)	Returns year from date_expr as a number.
POWER(num_expr1, num_expr2)	Returns num_expr1 raised to num_expr2 power.
INT(num_expr)	Returns integer part of num_expr.
NUMVAL(char_expr) VAL(char_expr)	Converts char_expr to number. If char_expr is not a valid number, returns zero.
DATE() TODAY()	Returns today's date.
DATEVAL(char_expr)	Converts char_expr to a date.
CTOD(char_expr, fmt)	Converts char_expr to date format using fmt template.

Appendix B
Error Codes

This appendix describes the error codes returned by functions in the LabWindows/CVI SQL Toolkit. In many cases, you can obtain additional information about errors by using `DBErrorMessage`.

Table B-1. Error Codes

Error Code	Description/Cause
1100	Error on menu operation. Resources may be getting low.
1500	Not enough memory for data transfer. Message truncated.
1501	Cannot create specified file. Use `DBErrorMessage` for more information.
1502	Cannot delete specified file. Use `DBErrorMessage` for more information.
1503	Not enough memory for this command.
1504	Cannot set current working directory to specified directory. Use `DBErrorMessage` for more information.
1506	Insufficient disk space.
1507	Invalid file handle.
1508	Access denied to specified file. Use `DBErrorMessage` for more information.
1509	Specified file not found. Use `DBErrorMessage` for more information.
1510	Specified path not found. Use `DBErrorMessage` for more information.
1511	You must run SHARE when locking is enabled or you must set Locking=NONE in your `ODBC.INI` file.
1512	Part or all of the region is already locked.
1513	Unable to unlock specified record.
1514	Lock failed because SHARE buffers exceeded.
2100	You can only login to this database once.
2105	Cannot load specified dynamic link library. Use `DBErrorMessage` for more information.
2106	Specified connection string is missing DSN=<driver name>. Use `DBErrorMessage` for more information.
2108	This Database driver does not support transaction processing.
2700	Specified token is too big. Use `DBErrorMessage` for more information.
2701	Specified number is too large. Use `DBErrorMessage` for more information.

(continues)

Table B-1. Error Codes (Continued)

Error Code	Description/Cause
2702	Invalid character in number. Use `DBErrorMessage` for more information.
2703	Unmatched quote character. Use `DBErrorMessage` for more information.
2704	Error parsing connect string. Use `DBErrorMessage` for more information.
2705	Error parsing string. Use `DBErrorMessage` for more information.
2706	Attribute specified more than once. Use `DBErrorMessage` for more information.
2707	Attribute specified twice by different keywords. Use `DBErrorMessage` for more information.
2708	Invalid hex character during conversion.
2709	Quicksort stack overflow.
2710	Too many sort keys.
2711	Invalid specified license file. Use `DBErrorMessage` for more information.
2712	Beta period expired.
2713	Evaluation period expired.
2714	Beta period will expire in less than 15 days.
2715	Evaluation period will expire in less than 15 days.
2716	String too large. Limit is 65,500 bytes.
2717	Initialization file is not open.
2718	Not for resale version.
2719	Could not create trace window.
3501	Format mask too long. Limit is 79 characters.
3502	A sign character must follow the E format character (for example, 0.00E+00).
3503	One or more digits for the exponent must follow the E format character (for example, 0.00E+00).
3504	Quoted string in format mask is missing closing quote.
3505	* for multiply or / for divide must follow the scale command (for example, [S*1000]).

(continues)

Table B-1. Error Codes (Continued)

Error Code	Description/Cause
3506	Power of 10 must follow the scale command (for example, [S*10]).
3507	A command in format mask is missing the end command character ‘]’.
3508	Partial values cannot be formatted or converted.
3509	Attempt to format or convert an invalid date value.
3510	Overflow when converting to single precision floating point.
3511	Overflow when converting to short integer.
3512	Overflow when converting to binary coded decimal.
3513	Exponent too large when converting value.
3514	Overflow when converting to long integer.
3515	Invalid year in date.
3516	Invalid month in date.
3517	Invalid day in date.
3518	Invalid hour in date.
3519	Invalid minute in date.
3520	Invalid second in date.
3521	Invalid fractional seconds in date.
3522	Invalid character in date format string. Use `DBErrorMessage` for more information.
3523	Invalid character in number format string. Use `DBErrorMessage` for more information.
3524	Invalid character in general format string. Use `DBErrorMessage` for more information.
3525	Specified string cannot be converted to a number. Use `DBErrorMessage` for more information.
3526	Could not convert date to value. Use `DBErrorMessage` for more information.
3527	Overflow when converting to double precision floating point.
4100	Limit on number of connection and statement handles exceeded.
4101	Invalid connection, statement, or query handle.
4102	LIKE or NOT LIKE invalid for non-character data type.
4103	Invalid column number.

(continues)

Table B-1. Error Codes (Continued)

Error Code	Description/Cause
4104	Column information requested does not apply to column because of column data type.
4105	Too many active programs using the toolkit.
4106	Columns must be retrieved in increasing order. The value of a column can only be retrieved once.
4110	Maximum length parameter of `DBBindColToChar` and `DBMapColToChar` must be zero if the underlying data type is not a character string.
4111	Attempt to bind column after records have been fetched.
4112	All columns in the `SELECT` statement must be bound to variables.
4114	Connected database does not support transactions.
4117	Attempt to commit or rollback a transaction without beginning a transaction.
4118	Transaction already active.
4119	Attempt to execute SQL statement without specifying SQL statement.
4120	Tracing already active.
4121	Invalid trace filename.
4122	Tracing is not active.
4123	Attempt to call `DBAllowFetchAnyDirection` after binding variables or fetching records.
4125	Attempt to fetch previous or random record without enabling fetching in any direction.
4127	Function is not valid for non-SELECT statements.
4128	Evaluation copy expired.
4129	Function is not valid if there is no active record.
4130	Evaluation copy will expire within 15 days.
4131	You cannot change this column’s value.
4132	Attempt to get column attribute that does not exist for this table.
4133	Dictionary query is not allowed for this function.
4134	Invalid option specified. Use `DBErrorMessage` for more information.
4135	Statement has not been executed or is not positioned on a row.
4136	Row to be locked has changed.
4137	Multiple rows locked.

(continues)

Table B-1. Error Codes (Continued)

Error Code	Description/Cause
4138	No rows locked.
4139	Specified column is not searchable.
4140	No database source specified.
4141	Invalid specified parameter number. Use `DBErrorMessage` for more information.
4142	Specified field number is invalid. Use `DBErrorMessage` for more information.
4143	Missing key word. Use `DBErrorMessage` for more information.
4144	Specified statement has not been executed. Execution required for this operation.
4145	`DBBeginTran` required before locking records.
4146	No query save file specified.
4147	Cannot insert row. For this database system, the row must be within the fetched rows or immediately following the last row fetched. Use `DBErrorMessage` for more information.
4148	Parameter type not in range 1 to 6. Use `DBErrorMessage` for more information.
4149	Native database error. Use `DBErrorMessage` for more information.
4150	Native database warning. Use `DBErrorMessage` for more information.
4151	Statement handle was not invalidated at end of transaction.
4152	Not positioned on any row.
4153	Attempt to fetch before committing or rolling back transaction.
4154	Query does not have a valid connection handle.
4155	Operation only allowed with deferred auto-update.
4500	Missing key word. Use `DBErrorMessage` for more information.
4501	Unexpected text at end of SQL query. Use `DBErrorMessage` for more information.
4502	Empty SQL clause.
4503	Missing ‘*/’ in comment.
4504	Improper select list in SELECT statement. Use `DBErrorMessage` for more information.
4505	No SQL statement given for execution.

(continues)

Table B-1. Error Codes (Continued)

Error Code	Description/Cause
4506	Cannot update or delete record, no primary key.
4507	Operation aborted.
4508	Duplicate table names in FROM clause. Use `DBErrorMessage` for more information.
4510	Unable to lock record. Modified or deleted by another user.
4511	Unable to insert record into database.
4513	Number of parameters supplied does not match number of parameter markers in statement.
4514	Declared parameter names do not match the statement parameters.
4515	Attempt to delete the current record from a query containing a join.
4516	Attempt to insert a record into a query containing a join.
4517	Another break cannot be added to the specified field. Use `DBErrorMessage` for more information.
4518	Attempt to read backwards without a log file.
4519	Unable to build `SELECT` list.
4520	Attempt to modify read-only query.
4522	Specified field number is too large. Use `DBErrorMessage` for more information.
4523	Case-insensitive search requested for a non-character column.
4524	Operation requires an executed statement..
4525	Internal Error, invalid ODBC handle.
4526	Table or table alias name exceeds limit. Use `DBErrorMessage` for more information.
4527	The specified parameter number is too large. Use `DBErrorMessage` for more information.
4528	At least one parameter has not been supplied a value.
4529	Fixing bind and set is not allowed for multiple-value parameters.
4530	End of results.
4532	Fetching no longer allowed on this statement probably due to previous update or delete. Enabling logging will probably fix the error.
4534	Attempt to change the data type of a parameter. Use `DBErrorMessage` for more information.

(continues)

Table B-1. Error Codes (Continued)

Error Code	Description/Cause
4537	Inserting specified row is illegal with current row. Use `DBErrorMessage` for more information.
4538	Attempt to insert too many characters into a column. Use `DBErrorMessage` for more information.
10028	Cannot access drive.
30040	Cannot open file. Use `DBErrorMessage` for more information.
30041	Error during file I/O.
30042	Cannot rename file. Use `DBErrorMessage` for more information.
30043	Not enough memory for this command.
30045	The maximum number of files are already open.
30047	Reserved file name cannot be opened. Use `DBErrorMessage` for more information.
30190	Out of memory.

Appendix C
Format Strings

This appendix describes the format strings that you can use with `DBMapColumnToChar` and `DBBindColChar`.

Format Strings

Format strings consist of symbols that describe how a value should be formatted. Table C-1 shows some example format strings. The symbols used in these examples are described later in this appendix.

Table C-1. Example Format Strings

Format String	Value	Formatted Value
mm/dd/yy	Mar 14, 1995	03/14/95
dd.mm.yy	Mar 14, 1995	14.03.95
‘Stephen Hawkins, born’ Mmmm d, yyyy	Mar 14, 1995	Stephen Hawkins, born March 14, 1995
hh:mm:ss	3:47:42 PM	15:47:42
hh:mm:ss AM/PM	3:47:42 PM	03:47:42 PM
$#,##0.00	210.6	$210.60
$#,##0.00;($#,##0.00)	210.6	$210.60
	-156.20348	($156.20)
GN	153	153
	1.875	1.875
0[S/1000]	12567	12
	199	0

Date/Time Format Strings

Date/time format strings control which parts of the date or time are converted or retrieved, the order of the parts, and how the months and days are abbreviated. Table C-2 lists the symbols you can use in date/time format strings.

Table C-2. Symbols for Date/Time Format Strings

Symbol	Description	Example Output
m	Month's number without leading zero.	12, 5
mm	Month's number with leading zero, if applicable.	12, 05
mmm	Month's three-letter abbreviation, lowercase.	mar
Mmm	Month's three-letter abbreviation, initial cap.	Mar
MMM	Month's three-letter abbreviation, uppercase.	MAR
mmmm	Month's full name, lowercase.	march
Mmmm	Month's full name, initial cap.	March
MMMM	Month's full name, uppercase.	MARCH
d	Day of the month's number without leading zero.	25, 5
dd	Day of the month's number with leading zero, if applicable.	25, 05
ddd	Day of the month's three-letter abbreviation, lowercase.	tue
Ddd	Day of the month's three-letter abbreviation, initial cap.	Tue
DDD	Day of the month's three-letter abbreviation, uppercase.	TUE
dddd	Day of the month's full name, lowercase.	tuesday
Dddd	Day of the month's full name, initial cap.	Tuesday
DDDD	Day of the month's full name, uppercase.	TUESDAY
yy	Last two digits of year.	60
yyyy	Four-digit year.	1960
h	Hour of the day, without leading zero (use am/pm symbol for 12-hour style).	12, 5
hh	Hour of the day, with leading zero (use am/pm symbol for 12-hour style).	12, 05
i (or m)	Minute of the hour, without leading zero.	57, 5
ii (or mm)	Minute of the hour, with leading zero.	57, 05

(continues)

Table C-2. Symbols for Date/Time Format Strings (Continued)

Symbol	Description	Example Output
s	Second of the minute, without leading zero.	57, 5
ss	Second of the minute, with leading zero.	57, 05
ss.ssssss	Second of the minute with fractional seconds (up to six ‘s’ symbols after the decimal point).	57.123456
am/pm	“am” or “pm” string, lowercase (forces 12-hour clock).	am
AM/PM	“AM” or “PM” string, uppercase (forces 12-hour clock).	AM
a/p	“a” or “p” string (forces 12-hour clock).	a
A/P	“A” or “P” string, uppercase (forces 12-hour clock).	A
J	Julian value for date/time.	
/-.:, <space>	Output the character.	
\<character>	Output the character following the ‘\’ character.	\G\M\T is GMT
“<string>“ ‘<string>‘	Output the string.	“GMT” is GMT
GD	General format for dates (the “Short Date Format” in the international section of the Windows control panel). *Do not combine other format symbols with GD except [US].*	
GDT	General format for dates with times (the “Time Format” in the international section of the Windows control panel is appended to the “Short Date Format”). *Do not combine other format symbols with GDT except [US].*	
GL	General long format for dates (the “Long Date Format” in the international section of the Windows control panel). *Do not combine other format symbols with GL except [US].*	
GLT	General long format for dates with times (the “Time Format” in the international section of the Windows control panel is appended to the “Long Date Format”). *Do not combine other format symbols with GLT except [US].*	
GT	General format for time. (the “Time Format” in the international section of the Windows control panel). *Do not combine other format symbols with GT.*	
[US]	Combine with GD, GDT, GL, GLT, GT to override the international section of the Windows control panel and use the United States defaults instead.	

Numeric Format Strings

You can use numeric format strings to format numbers in a variety of ways. Numeric formats can have either one or two sections separated by a semicolon. For formats with one section, use the same format for positive and negative numbers. For formats with two sections, use the second section as the format for negative numbers. Table C-3 lists the symbols you can use in numeric format strings.

Table C-3. Symbols for Numeric Format Strings

Symbol	Description
$	Outputs the currency string (from the international section of the Windows control panel).
.	Outputs the decimal point character (from the international section of the Windows control panel).
,	Outputs the thousands separator character (from the international section of the Windows control panel).
#	Outputs a digit. If there is no digit in the position, outputs nothing.
0	Outputs a digit. If there is no digit in the position, outputs a zero.
?	Outputs a digit. If there is no digit in the position, outputs a space.
%	Outputs the value as a percent. The value is multiplied by 100 and the '%' character is output.
e-	Outputs in scientific notation, shows exponent sign only if negative.
e+	Outputs in scientific notation, always shows exponent sign.
E+ E-	Outputs uppercase analogs of e+ and e-.
- + (), <space>	Outputs the character.
\<character>	Outputs the character following the '\' character.
"<string>"	Outputs the string.
'<string>'	
GN	General format for numbers. This is the default if no format string is given. *Note you can only combine GN with symbols that are enclosed in brackets, such as [US].*
GF	General fixed format for numbers (from the international section of the Windows control panel). *Note you can only combine GF with symbols that are enclosed in brackets, such as [US].*

(continues)

Table C-3. Symbols for Numeric Format Strings (Continued)

Symbol	Description
GC	General currency format for numbers (from the international section of the Windows control panel). *Note you can only combine GC with symbols that are enclosed in brackets, such as [US]*
[S/n]	Scales (divides) the number by a power of 10 before output. n must be a power of 10.
[S*n]	Scales (multiplies) the number by a power of 10 before output. n must be a power of 10.
[US]	Ignores the information in the international section of the Windows control panel. Substitutes the United States defaults instead.

Appendix D
Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help National Instruments solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *LabWindows/CVI Technical Support Form* before contacting National Instruments enables us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, you can reach our applications engineers Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

Corporate Headquarters
(512) 795-8248
Technical support fax: (800) 328-2203
(512) 794-5678

Branch Offices	**Phone Number**	**Fax Number**
Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	519 622 9311
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 71 11
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 48301892	02 48301915
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 202 2544	5 520 3282
Netherlands	03480 33466	03480 30673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 20 51 51	056 20 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

LabWindows®/CVI Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware. Use your completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

Be sure to fax copies of your `AUTOEXEC.BAT` and `CONFIG.SYS` files as well. If one or more National Instruments hardware products are involved in this problem, include the Hardware Configuration form from each hardware product's user manual. Include additional pages as necessary.

Name _____

Company _____

Address _____

_____

Fax (_____) _____ Phone (_____) _____

Computer brand _____ Model _____ Processor _____ Coprocessor _____

Operating system _____ Version _____ Bus (XT/AT/ISA, Micro Channel, or EISA) _____

Speed (MHz) CPU_____ BUS____ RAM _____ (Extended) _____ (Expanded) _____

Video Board _____ Mouse (Yes/No) Mouse Type _____ Mouse Driver Version _____

Other adapters installed _____

Base I/O Address Level of Other Boards _____ Interrupt Level of Other Boards _____

Hard disk capacity _____ Brand _____

Instruments used _____

National Instruments hardware product models _____ Revision _____

Configuration _____

Base I/O Address of Board(s) _____ Interrupt Level of Board(s) _____

LabWindows/CVI Version Number _____ Size and date of CVI.EXE file _____

Other National Instruments software product _____ Version _____

Programming Language and Version _____

The problem is _____

_____

List any error messages _____

_____

_____

The following steps will reproduce the problem _____

_____

_____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **LabWindows®/CVI SQL Toolkit Reference Manual**

Edition Date: **May 1995**

Part Number: **320960A-01**

Please comment on the completeness, clarity, and organization of this manual.

If you find errors in this manual, please record the page numbers and describe the errors.

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

Phone (_____) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway, MS 53-02
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
MS 53-02
(512) 794-5678

Index

A

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W