LabWindows/CVI SQL Toolkit Reference Manual

for Windows 95/98/NT

NATIONAL INSTRUMENTS™

September 1998 Edition
Part Number 320960B-01

Internet Support
E-mail: `support@natinst.com`
FTP Site: `ftp.natinst.com`
Web Address: `http://www.natinst.com`

Bulletin Board Support
BBS United States: 512 794 5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support
512 418 1111

Telephone Support (USA)
Tel: 512 795 8248
Fax: 512 794 5678

International Offices
Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11,
France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091,
Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00,
Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200,
United Kingdom 01635 523545

National Instruments Corporate Headquarters
6504 Bridge Point Parkway Austin, Texas 78730-5039 USA Tel: 512 794 0100

© Copyright 1995, 1998 National Instruments Corporation. All rights reserved.

Important Information

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

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

CVI™, `natinst.com`™, National Instruments™, and the National Instruments logo are trademarks of National Instruments Corporation.

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

About This Manual

Organization of This Manual xi
Conventions Used in This Manual xii
Related Documentation xiii
Customer Communication xiii

Chapter 1
Introduction

Installation 1-1
Installing the LabWindows/CVI SQL Toolkit Software 1-1
Overview 1-3

Chapter 2
Getting Started

Database Concepts 2-1
Structured Query Language (SQL) 2-2
ODBC Standard 2-3
ODBC Administrator 2-3
Third-Party ODBC Database Drivers 2-4
ADO Providers 2-4
Database Sessions 2-5
Step 1. Connecting to the Database 2-5
Step 2. Activating SQL Statements 2-5
Step 3. Processing SQL Statements 2-5
Step 4. Deactivating SQL Statements 2-6
Step 5. Disconnecting from the Database 2-6

Chapter 3
Using the SQL Toolkit Library

Function Summary 3-1
Connecting to a Database 3-3
Using Automatic SQL (Maps) 3-3
Using Explicit SQL Statements 3-5
Fetching Records 3-6
Inserting a Record 3-7
Updating a Record 3-8
Deleting a Record 3-8

Deleting a Table 3-9
Information Functions 3-9
Data Source Information 3-9
SELECT Statement Information 3-10
Transactions 3-11
Error Checking 3-12

Chapter 4
Advanced Operations

Setting Connection Attributes 4-1
Setting Statement Attributes 4-2
Working with Compound Statements 4-2
Parameterized Statements 4-2
Alternative Data Access Methods 4-4
Compatibility with SQL Toolkit 1.1 4-6
Multithreaded Support 4-6

Chapter 5
SQL Toolkit Library Reference

SQL Toolkit Library Function Overview 5-1
SQL Toolkit Library Function Reference 5-4
DBActivateMap 5-5
DBActivateSQL 5-7
DBAllowFetchAnyDirection 5-9
DBBeginMap 5-10
DBBeginTran 5-12
DBBindColBinary 5-14
DBBindColChar 5-16
DBBindColDouble 5-19
DBBindColFloat 5-21
DBBindColInt 5-23
DBBindColShort 5-25
DBCancelRecordChanges 5-27
DBClearParam 5-29
DBCloseConnection 5-30
DBClosePreparedSQL 5-31
DBCloseSQLStatement 5-33
DBColumnName 5-35
DBColumnType 5-36
DBColumnWidth 5-38
DBCommit 5-39
DBConnect 5-40

DBCreateParamBinary 5-42
DBCreateParamChar 5-45
DBCreateParamDouble 5-48
DBCreateParamFloat 5-51
DBCreateParamInt 5-53
DBCreateParamShort 5-56
DBCreateRecord 5-59
DBCreateTableFromMap 5-61
DBDatabases 5-63
DBDeactivateMap 5-65
DBDeactivateSQL 5-66
DBDeleteRecord 5-67
DBDiscardConnection 5-68
DBDiscardSQLStatement 5-69
DBDisconnect 5-70
DBError 5-71
DBErrorMessage 5-72
DBExecutePreparedSQL 5-73
DBFetchNext 5-75
DBFetchPrev 5-76
DBFetchRandom 5-78
DBForeignKeys 5-80
DBFree 5-83
DBFreeVariantArray 5-84
DBGetColBinary 5-86
DBGetColBinaryBuffer 5-88
DBGetColChar 5-90
DBGetColCharBuffer 5-92
DBGetColDouble 5-94
DBGetColFloat 5-96
DBGetColInt 5-98
DBGetColShort 5-100
DBGetColumnAttribute 5-102
DBGetColVariant 5-106
DBGetConnectionAttribute 5-108
DBGetParamAttribute 5-114
DBGetParamBinary 5-118
DBGetParamBinaryBuffer 5-121
DBGetParamChar 5-124
DBGetParamCharBuffer 5-126
DBGetParamDouble 5-129
DBGetParamFloat 5-131
DBGetParamInt 5-133

DBGetParamShort 5-135
DBGetParamVariant 5-137
DBGetSQLToolkitVersion 5-139
DBGetStatementAttribute 5-140
DBGetVariantArray 5-149
DBGetVariantArrayColumn 5-151
DBGetVariantArrayValue 5-155
DBImmediateSQL 5-158
DBIndexes 5-160
DBInit 5-163
DBMapColumnToBinary 5-164
DBMapColumnToChar 5-167
DBMapColumnToDouble 5-170
DBMapColumnToFloat 5-173
DBMapColumnToInt 5-176
DBMapColumnToShort 5-179
DBMoreResults 5-182
DBNativeError 5-184
DBNewConnection 5-185
DBNewSQLStatement 5-186
DBNumberOfColumns 5-188
DBNumberOfModifiedRecords 5-189
DBNumberOfRecords 5-190
DBOpenConnection 5-191
DBOpenSchema 5-192
DBOpenSQLStatement 5-198
DBPrepareSQL 5-199
DBPrimaryKeys 5-202
DBPutColBinary 5-204
DBPutColChar 5-206
DBPutColDouble 5-208
DBPutColFloat 5-210
DBPutColInt 5-212
DBPutColNull 5-214
DBPutColShort 5-216
DBPutColVariant 5-218
DBPutRecord 5-220
DBRefreshParams 5-222
DBRollback 5-224
DBSetAttributeDefault 5-225
DBSetBackwardCompatibility 5-228
DBSetColumnAttribute 5-229
DBSetConnectionAttribute 5-231

DBSetDatabase 5-236
DBSetParamAttribute 5-237
DBSetParamBinary 5-241
DBSetParamChar 5-243
DBSetParamDouble 5-245
DBSetParamFloat 5-247
DBSetParamInt 5-249
DBSetParamShort 5-251
DBSetParamVariant 5-253
DBSetStatementAttribute 5-255
DBSources 5-262
DBTables 5-264
DBUpdateBatch 5-267
DBWarning 5-269

Appendix A
SQL Language Reference

SQL Commands A-1
SQL Objects A-2
SQL Clauses A-3
SQL Operators A-4
SQL Functions A-6

Appendix B
Error Codes

Appendix C
Format Strings

Format Strings C-1
Date/Time Format Strings C-2
Numeric Format Strings C-5

Appendix D
Customer Communication

Glossary

Index

Figures

Figure 1-1. LabWindows/CVI SQL Toolkit Installation Dialog Box 1-2

Figure 2-1. Data Sources Dialog Box 2-3
Figure 2-2. ODBC dBASE Driver Setup Dialog Box 2-4

Figure 3-1. Summary of SQL Library Functions 3-2

Tables

Table 2-1. Sample Test Sequence Results 2-1
Table 2-2. Data Types Supported by the Lab/Windows/CVI SQL Toolkit 2-2

Table 3-1. Sample Database Table 3-3

Table 5-1. Function Tree for the SQL Toolkit Library 5-1
Table 5-2. Functions for Database Transactions 5-12
Table 5-3. Query Types 5-193

Table A-1. SQL Commands A-1
Table A-2. SQL Objects A-2
Table A-3. SQL Clauses A-3
Table A-4. SQL Operators A-4
Table A-5. SQL Functions A-6

Table B-1. SQL Toolkit Error Code B-1
Table B-2. Error Codes for CVI Automation B-5
Table B-3. Error Codes for OLE B-5

Table C-1. Example Format Strings C-1
Table C-2. Symbols for Date/Time Format Strings C-2
Table C-3. Symbols for Numeric Format Strings C-5

About This Manual

The *LabWindows/CVI SQL Toolkit Reference Manual* describes this LabWindows/CVI add-on package which you can use to perform 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 LabWindows/CVI 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 Open Database Connectivity (ODBC) standard, and the Database Session.
- Chapter 3, *Using the SQL Toolkit Library*, describes how to use the functions in the LabWindows/CVI SQL Library for common types of database operations and contains example code for performing each operation.
- Chapter 4, *Advanced Operations*, describes how to use the SQL Library functions for advanced types of database operations and contains example code for performing each operation.
- Chapter 5, *SQL Toolkit Library Reference*, describes each function in the LabWindows/CVI SQL Toolkit. The functions appear in alphabetical order, with a description of the function and its C syntax, a description of each parameter, and a list of possible error codes.
- Appendix A, *SQL Language 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 you can use to request help from National Instruments or to comment on our products and documentation.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this manual.

Conventions Used in This Manual

The following conventions appear in this manual:

<> — Angle brackets enclose the name of a key on the keyboard, for example, <shift>.

- — A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Ctrl-Alt-Delete>.

» — The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options»Substitute Fonts** directs you to pull down the **File** menu, select the **Page Setup** item, select **Options**, and finally select the **Substitute Fonts** options from the last dialog box.

☞ — This icon to the left of bold italicized text denotes a note, which alerts you to important information.

bold — Bold text denotes the names of menus, menu items, parameters, dialog boxes, dialog box buttons or options, icons, windows, Windows 95 tabs, or LEDs.

bold italic — Bold italic text indicates a note.

italic — Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.

`monospace` — Text in this font denotes text or characters that you should literally enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and for code excerpts.

`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, and files.

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. These forms are in Appendix D, *Customer Communication*, at the end of this manual.

1

Introduction

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

Installation

This section contains instructions for installing the LabWindows/CVI SQL Toolkit on the Windows 95/NT platform.

Installing the LabWindows/CVI 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:

- Select **Run** from the **Start** menu in the Windows taskbar. In the dialog box that appears, enter *x*`:\setup`, where *x* denotes the proper letter designation of the drive. Click on **OK** and follow the instructions that appear on your screen.
- Launch the Windows Explorer. Click on the icon of the drive 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.

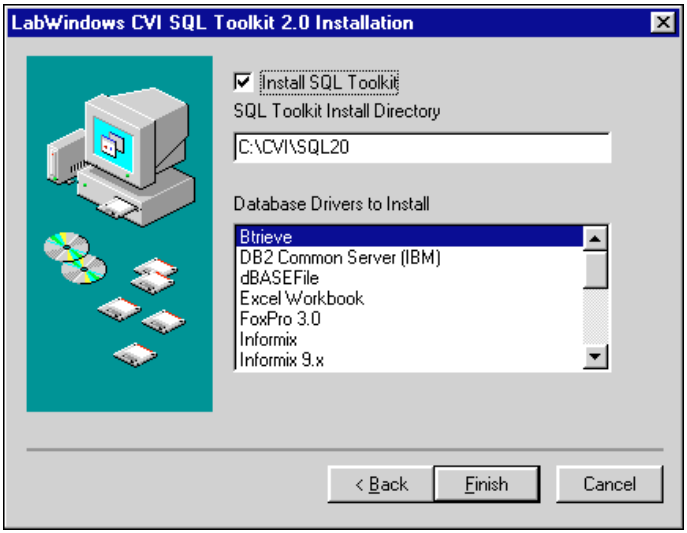

Figure 1-1. LabWindows/CVI SQL Toolkit Installation Dialog Box

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

- **SQL Toolkit**—Contains library file function panels and LabWindows/CVI example programs for database communication. The default installation directory for the toolkit files is the `\sql20` subdirectory within the LabWindows/CVI base directory.
- **ODBC Database Drivers**—Contains database-specific DLLs required to communicate with each database. LabWindows/CVI installs the ODBC Database Driver files in the `\windows\system` directory under Windows 95/98 or in the `\winnt\system32` directory under Windows NT.
- **ADO Support**—Contains the files that implement the Microsoft Active Data Object (ADO). The ADO installer places ADO files in `\program files\common files\system\ado`.

In addition, the driver setup program modifies or creates `control.ini`, `odbc.ini`, and `odbcinst.ini` in the registry.

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 and advanced functions for customized tasks.

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

- Works with any provider that adheres to the Microsoft Active Data Object (ADO) standard.
- Works with any database driver that complies with ODBC.
- Maintains a high level of portability. In many cases, you can port your application to another database by changing the connection string you pass to the `DBConnect` function.
- Converts database column values from native data types to standard LabWindows/CVI SQL Toolkit data types, further enhancing portability.
- The default ADO ODBC provider 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.
- Creates tables and selects, inserts, updates, and deletes records without using SQL statements.

Because of the wide range of databases the LabWindows/CVI SQL Toolkit works with, some portability issues remain. You should consider the following issues 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 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 pass the upper or lower limits of the precision available for a BCD value.
- Restrictions on column names vary among database systems. For maximum portability, limit column names to 10 uppercase characters without spaces.
- Some database systems do not support date, time, or date and time data types.

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 records, also known as rows, and fields, also known as columns. Every table in a database must have a unique name. Similarly, every field 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. This manual refers to `NULL` values in tables as SQL Null values, to distinguish them from standard `NULL` values.

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 toolkit automatically maps these data types into the appropriate type in the underlying database. By using the common data types, the toolkit program can access a variety of databases with little or no modification. Table 2-2 lists the data types that the toolkit supports.

Table 2-2. Data Types Supported by the Lab/Windows/CVI 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.

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 Language Reference*, for a complete list of SQL commands.

ODBC Standard

The SQL Access Group—including representatives of Microsoft, Tandem, Oracle, Informix, and Digital Equipment Corporations—developed the Open Database Connectivity (ODBC) standard as a uniform method for applications to access databases. ODBC 1.0 was released in September 1992. The standard includes a driver packaging standard, a method for maintaining Data Source Names, and a SQL implementation based on ANSI SQL. LabWindows/CVI SQL Toolkit currently supports ODBC version 3.5. Because the toolkit and the drivers that come with it comply with the ODBC standard, you can port LabWindows/CVI database applications to other supported databases with minimal changes.

ODBC Administrator

The Open Database Connectivity (ODBC) standard depends on ODBC drivers for each database system you access. You must register any ODBC driver that you use. You can use the ODBC Administrator icon on your Control Panel to register and configure drivers to make them available as data sources for your applications. Your system saves all changes you make within the ODBC Administrator.

When you double-click on the ODBC Administrator icon on the Control Panel the ODBC Data Source Administrator dialog box appears, as shown in Figure 2-1.

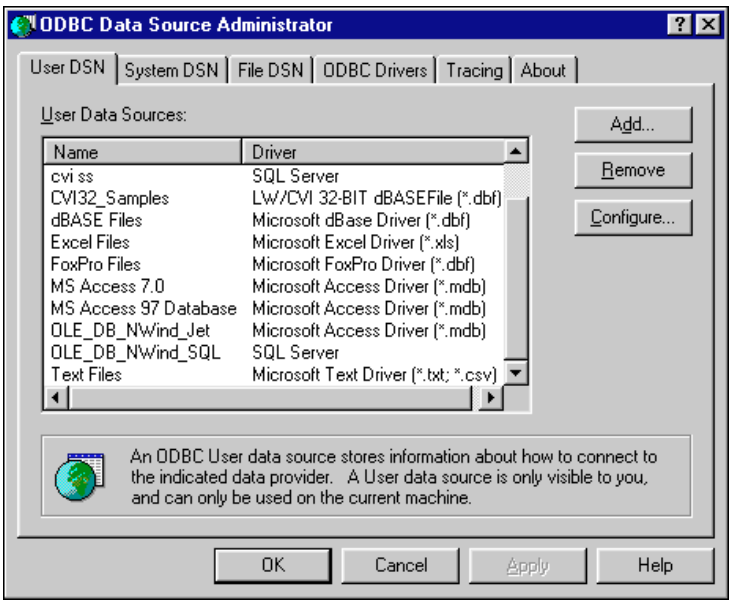

Figure 2-1. Data Sources Dialog Box

The ODBC Data Source Administrator dialog box lists all the registered ODBC data sources. You can use the **Add** or **Configure** buttons to display a driver-specific dialog box where you can configure a new or an existing data source. The system then saves the configuration for the data source in the Registry.

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

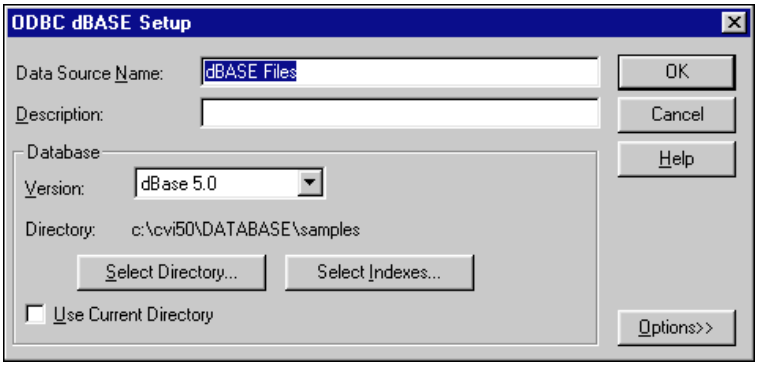

Figure 2-2. ODBC dBASE Driver Setup Dialog Box

Third-Party ODBC Database Drivers

The LabWindows/CVI SQL Toolkit supplies several ODBC database drivers. Database system vendors and third-party developers offer more powerful version of these drivers. They also offer drivers for other database systems. The LabWindows/CVI SQL Toolkit complies with the ODBC standard, so you can use it with any ODBC-compliant database drivers. Please refer to your vendor documentation for information on registering your specific database drivers with the ODBC Administrator.

ADO Providers

In addition to ODBC data sources, you can also access data through any provider that complies with the Microsoft Active Data Object (ADO) standard. All data access in the LabWindows/CVI SQL Toolkit occurs through an ADO provider. If you do not specify a provider, the toolkit automatically uses the default ODBC provider described above. Remember that ADO providers are not required to support SQL; some use their own command language.

Database Sessions

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.

Step 1. Connecting to the Database

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

Step 2. Activating SQL Statements

With the LabWindows/CVI SQL Toolkit, you can use several methods for activating statements, such as 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.
- Advanced functions give you more control over statement execution.

For more details on automatic SQL and explicit SQL, refer to Chapter 3, *Using the SQL Toolkit Library*. For more details on advanced functions, refer to Chapter 4, *Advanced Operations*.

Step 3. Processing SQL Statements

In general, only SQL `SELECT` statements require further processing. `SELECT` statements are important components of the LabWindows/CVI 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 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 Library*.

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.

Step 4. 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.

Step 5. Disconnecting from the Database

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

3

Using the SQL Toolkit Library

This chapter describes how to use the functions in the LabWindows/CVI SQL Library for common types of database operations and contains example code for performing each operation.

Function Summary

Figure 3-1 shows how the major SQL Library functions relate to each other. The remainder of this chapter describes the steps shown in the following illustration.

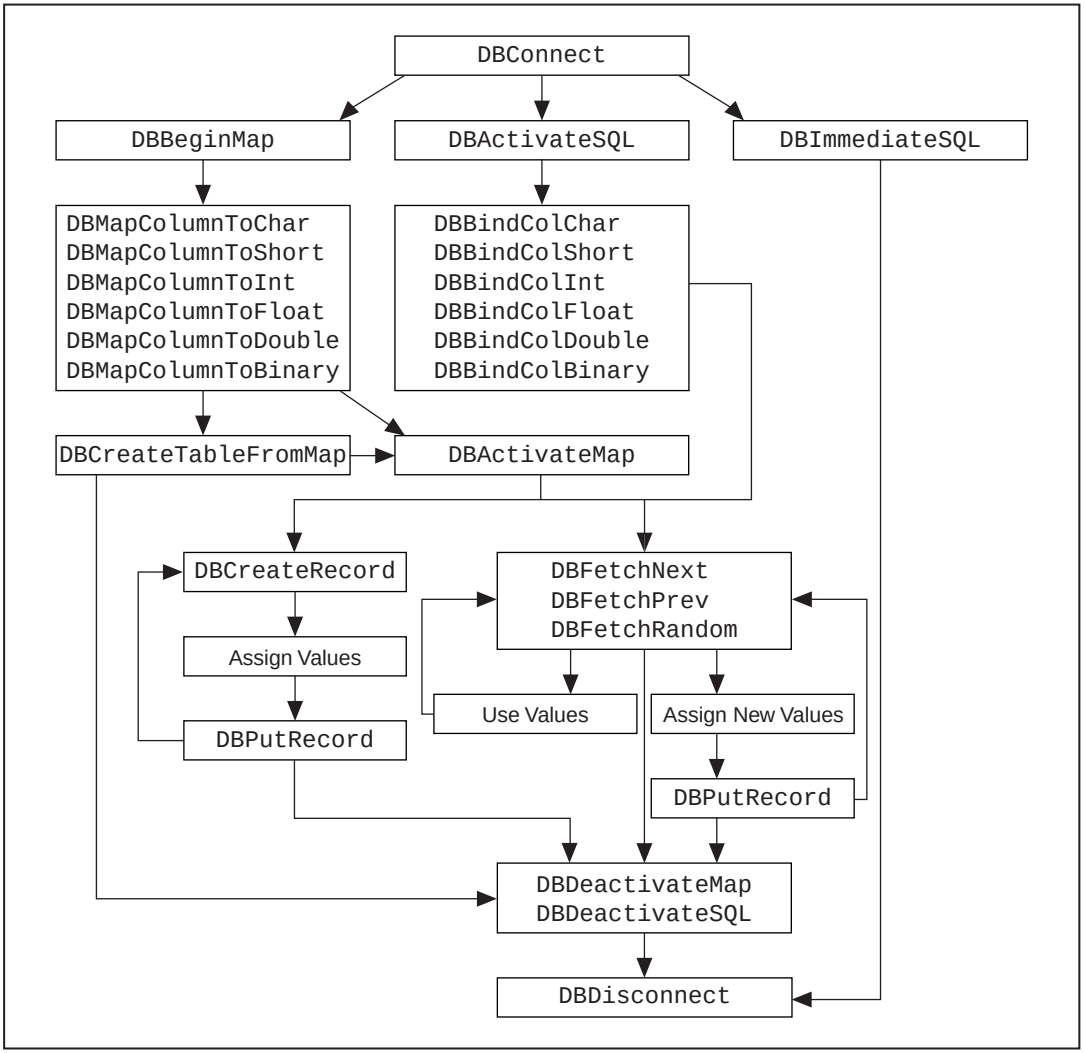

Figure 3-1. Summary of SQL Library Functions

The examples in the remainder of this chapter refer to the data in Table 3-1.

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 Library 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 use the `DBMapColumnTo` functions (in this case, `DBMapColumnToChar` and `DBMapColumnToDouble`) to map the columns you want to read into program variables. All the `DBMapColumnTo` functions use the following parameters: map handle, column name, address of the target variable, and address of the status variable. `DBMapColumnToChar` has two additional parameters: 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 map 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 Library 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. `DBCreateTableFromMap` uses the same parameters that `DBActivateMap` uses: map handle and table name. After creating the table, you can continue to use the map with `DBActivateMap`. For example, you can use the map to create the initial records for the table. The following example code appears 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 an SQL `SELECT` statement. Notice that the supplied `SELECT` statement contains a `WHERE` clause, which is not possible when you use 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. When you finish 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;
```

Most SQL statements other than `SELECT` do not require further processing. For this reason, 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

Automatic SQL and explicit SQL require the same functions for fetching records. Typically, you fetch records from first to last with `DBFetchNext`. The parameter for `DBFetchNext` consists of the statement handle from `DBActivateMap` 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 excerpt 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 that `DBFetchNext` requires, except that you must also pass the record number to `DBFetchRandom`. You cannot use `DBFetchPrev` or `DBFetchRandom` if the statement uses a forward-only cursor. The following example first fetches the last record (notice the use of `DBNumberOfRecords`) and then fetches the remaining records in reverse order. This code excerpt comes from the `readtab3.prj` project.

```
/* Use bi-directional cursor. */
resCode = DBSetAttributeDefault (hdbc, ATTR_DB_CURSOR_TYPE,
                                 DB_CURSOR_TYPE_KEYSET);
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 using the same methods you use to fetch 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 both automatic SQL and 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 you activate a map or SQL statement, you must 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 use SQL statements to delete a table.***

Information Functions

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

Data Source Information

The 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. The SELECT statement that DBSources executes returns two columns: the source name 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 returns no records.
- DBTables returns information about the available tables. Its parameters contain the connection handle, a table catalog, a table schema, a table name, and a flags parameter. The flags parameter determines the type of table. The SELECT statement returns the table catalog, table schema, table name, table type, table GUID, and description.

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, cat, &catStat, "");
if (resCode != DB_SUCCESS) goto Error;
resCode = DBBindColChar(hstmt, 2, 127, schema, &schemaStat, "");
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 = DBBindColInt(hstmt, 5, &guid, &guidStat);
if (resCode != DB_SUCCESS) goto Error;
resCode = DBBindColChar(hstmt, 6, 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 Library 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 statement handle is the first parameter for all these functions. DBColumnName and DBColumnType also have a second parameter corresponding to the column number. DBNumberOfRecords, DBNumberOfColumns, and DBNumberOfModifiedRecords return the number of items in a table.

DBColumnName returns a pointer to the column name string. The toolkit reuses the buffer containing the column name. If you want to use this information, 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 call DBNumberOfRecords, so fetching in any direction */
/* must be enabled.  */
resCode = DBSetAttributeDefault (hdbc, ATTR_DB_CURSOR_TYPE,
DB_CURSOR_TYPE_KEYSET);
...
/* 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 LabWindows/CVI 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 a single 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 whether 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);
```

Note ***Some database drivers do not support transactions. Other drivers require that you start any transaction before activating any statements.***

Error Checking

The SQL Library 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 less than or equal to zero indicates an error. You can then call `DBError` to determine the error number. You also can call `DBErrorMessage` to get the text of the error message. For functions that return data, such as `DBColumnName`, you should call `DBError` to determine whether an error occurred.

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

4

Advanced Operations

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

Setting Connection Attributes

Sometimes, you need to set certain attributes of a connection such as the command timeout or the isolation level. You cannot set attributes when you use `DBConnect` because `DBConnect` creates and opens the connection in one step and most attributes must be set before the connection is opened. To allow setting connection attributes, the SQL Library provides advanced functions which allow you to create and open the connection separately. You can create a connection with `DBNewConnection` which has no parameters and returns a statement handle. You can then set the attributes with `DBSetConnectionAttribute`. The parameters of `DBSetConnectionAttribute` are the statement handle from `DBNewConnection`, the attribute to set and the value for the attribute. You can also use `DBGetConnectionAttribute` to query the value of connection attributes. After you set the attributes, call `DBOpenConnection` to open the connection.

In some cases, instead of closing and discarding a connection with `DBDisconnect`, you may wish to close a connection, change some attributes and reopen the connection. Use `DBCloseConnection` to close but not discard a connection. When you have finished using the connection, discard it with `DBDiscardConnection`.

```
int hdbc = 0;    /* Handle to database connection */
hdbc = DBNewConnection();
resCode = DBSetConnectionAttribute(hdbc,
          ATTR_DB_CONN_DEFAULT_DATABASE, "PUBS");
... /* set additional attributes */
resCode = DBOpenConnection(hdbc);
... /* use the connection */
resCode = DBCloseConnection (hdbc);
resCode = DBDiscardConnection (hdbc);
```

Setting Statement Attributes

Like connection attributes, most statement attributes cannot be set once the statement has been executed. To set statement attributes you must use DBNewSQLStatement and DBOpenSQLStatement (or DBPrepareSQL and DBExecutePreparedSQL which are discussed in the next section). DBNewSQLStatement takes two parameters, a connection handle and the SQL statement and returns a statement handle. DBOpenSQLStatement has one parameter, the statement handle from DBNewSQLStatement. After you create a statement you can call DBSetStatementAttribute to set the statement attributes.

Like a connection, a statement can also be reused if you close it, but do not discard it. Use DBCloseSQLStatement to close a statement and DBDiscardSQLStatement to discard a statement.

```
hstmt = DBNewSQLStatement (hdbc, "SELECT UUT_NUM, MEAS1,
                          MEAS2 FROM TESTRES");
resCode = DBSetStatementAttribute(hstmt,
                                  ATTR_DB_STMT_CURSOR_LOCATION,
                                  DB_CURSOR_LOC_SERVER);
resCode = DBOpenSQLStatement(hstmt);
... /* process the statement */
resCode = DBCloseSQLStatement (hstmt);
hstmt = DBDiscardSQLStatement(hstmt);
```

Working with Compound Statements

Some database systems support compound SQL statements like SELECT * FROM TESTRES; SELECT * FROM PREVRES. You can call DBMoreResults to allow fetching the results for the additional parts of the compound statement.

Parameterized Statements

Sometimes it is useful to have parameters in statements. Parameterized statements allow you to specify the SQL statement once, but vary the parameters (such as the matching criteria of a WHERE clause) over time.

To use parameters with a statement, first prepare the statement with DBPrepareSQL. After the statement is prepared, you can set statement attributes with DBSetStatementAttribute if required. Next you must create all the parameters using the DBCreateParam functions. All the DBCreateParam functions take at least four arguments, a statement handle, the name of the parameter (use "" if the parameter is not named), the parameter direction (in, out, in/out or return value) and the initial value (you can change the value later with the DBSetParam functions). DBCreateParamString and

DBCreateParamBinary have an additional argument, the maximum size. After creating all the parameters, execute the statement with DBExecutePreparedSQL. You can then fetch the data and perform whatever processing is needed. Use DBClosePreparedSQL and DBDiscardPreparedSQL to close and discard the statement.

```
hstmt = DBPrepareSQL (hdbc,
   "SELECT UUT_NUM, MEAS1, MEAS2 FROM TESTRES WHERE MEAS1 > ?");
resCode = DBCreateParamDouble(hstmt, "", DB_PARAM_INPUT, 1.0);
resCode = DBExecutePreparedSQL(hstmt);
... /*Process the results */
resCode = DBClosePreparedSQL(hstmt);
resCode = DBDiscardPreparedSQL(hstmt);
```

Some databases support stored procedures which also allow parameters. Using parameters with stored procedures is much like using parameters with statements with some additions. You must set the ATTR_DB_STMT_COMMAND_TYPE attribute to DB_COMMAND_STORED_PROC. If the stored procedure uses output parameters, the values will be incorrect until you close the statement.

```
/* This example works with SQL Server.        */
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest \
          ( @InParam int, @OutParam int OUTPUT ) as select \
          @OutParam = @InParam + 10 \
          SELECT * FROM Authors WHERE State <> 'CA' return
          @OutParam +10");
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_COMMAND_TYPE,
                                  DB_COMMAND_STORED_PROC);
resCode = DBCreateParamInt(hstmt, "", DB_PARAM_RETURN_VALUE, -1);
resCode = DBCreateParamInt(hstmt, "InParam", DB_PARAM_INPUT, 10);
resCode = DBCreateParamInt(hstmt, "OutParam", DB_PARAM_OUTPUT, -1);

/* Set the input parameter */
resCode = DBSetParamInt(hstmt, 1, 20);

/* Execute the statement */
resCode = DBExecutePreparedSQL(hstmt);
.../* fetch the returned data if any */

/* Close the statement to allow getting output parameters */
resCode = DBClosePreparedSQL(hstmt);

/* Get the values of the parameters */
resCode = DBGetParamInt(hstmt, 1, &retParam);
resCode = DBGetParamInt(hstmt, 2, &inParam);
resCode = DBGetParamInt(hstmt, 3, &outParam);
```

Alternative Data Access Methods

The DBGetCol and DBPutCol functions allow you to access data in the current record without binding the fields to program variables. Execution time for code using DBGetCol and DBPutCol functions is approximately the same as for code using the binding or mapping functions. You can also mix the two methods, for example, bind most columns, but use a DBGetCol function to get values for seldom used columns.

```
/* Execute a select statement */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, \
        LOOPNUM, MEAS1, MEAS2, CHANGER FROM REC1000");

/* Create a new record */
resCode = DBCreateRecord(hstmt);

/* Put values into the record. */
resCode = DBPutColInt(hstmt, 2, 47);
resCode = DBPutColDouble(hstmt, 4, 42.6);

/* Put the record to the database. */
resCode = DBPutRecord(hstmt);

/* Execute a select statement */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, CHANGER, \
        LOOPNUM, MEAS1, MEAS2 FROM REC1000");
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Get values into the record. */
    resCode = DBGetColChar(hstmt, 1, &uutNum);
    resCode = DBGetColInt(hstmt, 2, &changer);
    resCode = DBGetColFloat(hstmt, 3, &meas1);
    resCode = DBGetColDouble(hstmt, 4, &meas2);
    resCode = DBGetColShort(hstmt, 5, &loopNum);
    /* process values */
    ...
    DBFree(uutNum);
}
/* Deactivate the SQL statement */
```

You can also get all the data returned by a SELECT statement as one large array of variants using the DBGetVariantArray function. DBGetVariantArray is usually faster than fetching row by row, but is more complicated to use and uses much more memory.

The toolkit provides two functions for retrieving values from the array: DBGetVariantArrayValue obtains a single value and DBGetVariantArrayColumn obtains the values for a single field/column for a range of records. The following paragraphs give an example for each function.

The following example demonstrates use of `DBGetVariantArrayValue` with `DBGetVariantArray`:

```
VARIANT *cArray;
...
hstmt = DBActivateSQL (hdbc, "SELECT * FROM TESTRES");

resCode = DBGetVariantArray(hstmt, &cArray, &numRecs, &numFields);
for (i = 0; i < numRecs; i++) {
    for (j = 0; j < numFields; j++) {
        resCode = DBGetVariantArrayValue(cArray, numRecs,
                                         numFields, CAVT_CSTRING,
                                         i, j, &tempStr);
        if (resCode == DB_NULL_DATA){
            /* Handle null data. */
        } else {
            /* Handle other data. */
            DB_Free(tempStr);
        }
    }
}
resCode = DBFreeVariantArray(cArray, 1, numRecs, numFields);
```

The following example demonstrates use of `DBGetVariantArrayColumn` with `DBGetVariantArray`:

```
double *column;
...
hstmt = DBActivateSQL (hdbc, "SELECT * FROM TESTRES");

resCode = DBGetVariantArray(hstmt, &cArray, &numRecs, &numFields);
column = malloc(numRecs * sizeof(double));

resCode = DBGetVariantArrayColumn(cArray, numRecs, numFields,
                                  CAVT_DOUBLE, 1, 2, 3, column);
if (resCode == DB_NULL_DATA) {
    printf("Cannot process, some fields contain null\n");
} else {
    for(i = 0; i < 3; i++) {
           /* Process values.                                  */
           /* Note: Because the type of the values is not    */
           /* char* or BSTR, you should not free the values. */
        }
    }
    resCode = DBFreeVariantArray(cArray, 1, numRecs, numFields);
    free(column);
    hstmt = DBDeactivateSQL (hstmt);
```

Compatibility with SQL Toolkit 1.1

Version 2.0 of the LabWindows/CVI SQL Toolkit features greatly expanded capabilities compared to version 1.1. The following compatibility issues exist between the two versions:

- **Including cvi_db.h**—Version 2.0 programs must include `cvi_db.h` before `utility.h` and `formatio.h`.
- **Functions that Return Statement or Function Handles**—In version 1.1 these functions always returned zero when an error occurred. In version 2.0, these functions return error codes when an error occurs. You can use `DBSetBackwardCompatibility` to make the toolkit emulate the version 1.1 behavior.
- **Column Types**—In version 1.1, `DBColumnType` returned only eight types. In version 2.0, `DBColumnType` returns the full range of types available in ADO. You can use `DBSetBackwardCompatibility` to make the toolkit emulate the version 1.1 behavior.
- **Information Functions**—The set of columns that `DBSources`, `DBDatabases`, `DBTables`, `DBIndexes`, `DBPrimaryKeys`, and `DBForeignKeys` return is different for version 2.0. `DBSetBackwardCompatibility` does not change the columns these functions return; you must change any programs that use these functions.

Multithreaded Support

The SQL toolkit supports multithreaded applications. You must call `DBInit` to initialize multithreaded support. You do not have to call `DBInit` if your program is single threaded. Some ODBC drivers, especially drivers based on the Microsoft Jet database engine, do not work with multiple threads.

5

SQL Toolkit Library Reference

This chapter describes each function in the LabWindows/CVI SQL Toolkit. The functions appear in alphabetical order, with a description of the function and its C syntax, a description of each parameter, and a list of possible error codes.

SQL Toolkit Library Function Overview

This section contains general information about the SQL Toolkit Library functions and panels.

Table 5-1. Function Tree for the SQL Toolkit Library

Class/Panel Name	Function Name
Initialize Threading	`DBInit`
Connection	
Open New Connection	`DBConnect`
Set Default Attribute	`DBSetAttributeDefault`
Close and Discard 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`
Map Column to Binary	`DBMapColumnToBinary`
Create Table From Map	`DBCreateTableFromMap`
Activate Map	`DBActivateMap`
Deactivate Map	`DBDeactivateMap`
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`
Bind Column to Binary	`DBBindColBinary`
Deactivate SQL Statement	`DBDeactivateSQL`

Table 5-1. Function Tree for the SQL Toolkit Library (Continued)

Class/Panel Name	Function Name
Fetch Records	
Fetch Next Record	`DBFetchNext`
Fetch Previous Record	`DBFetchPrev`
Fetch Random Record	`DBFetchRandom`
Insert/Delete/Update Records	
Create New Record	`DBCreateRecord`
Delete Record	`DBDeleteRecord`
Put Record	`DBPutRecord`
Cancel Record Changes	`DBCancelRecordChanges`
Batch Update	`DBUpdateBatch`
Information Functions	
Data Source Information	
Available Sources	`DBSources`
Available Databases	`DBDatabases`
Available Tables	`DBTables`
Indexes Information	`DBIndexes`
Primary Keys Information	`DBPrimaryKeys`
Foreign Keys Information	`DBForeignKeys`
Open Schema	`DBOpenSchema`
SELECT Information	
Number of Records	`DBNumberOfRecords`
Number Of Columns	`DBNumberOfColumns`
Column Name	`DBColumnName`
Column Width	`DBColumnWidth`
Column Type	`DBColumnType`
Set Column Attribute	`DBSetColumnAttribute`
Get Column Attribute	`DBGetColumnAttribute`
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`
Freeing Resources	
Free Memory	`DBFree`
Compatibility	
Get Toolkit Version	`DBGetSQLToolkitVersion`
Set Backward Compatibility	`DBSetBackwardCompatibility`
Advanced Functions	
Advanced Connections	
New Connection	`DBNewConnection`

Table 5-1. Function Tree for the SQL Toolkit Library (Continued)

Class/Panel Name	Function Name
Set Connection Attribute	`DBSetConnectionAttribute`
Get Connection Attribute	`DBGetConnectionAttribute`
Open Connection	`DBOpenConnection`
Close Connection	`DBCloseConnection`
Discard Connection	`DBDiscardConnection`
Advanced Statements	
New SQL Statement	`DBNewSQLStatement`
Set Statement Attribute	`DBSetStatementAttribute`
Get Statement Attribute	`DBGetStatementAttribute`
Open SQL Statement	`DBOpenSQLStatement`
More Results	`DBMoreResults`
Get Records as Array	
Get Records as Array	`DBGetVariantArray`
Get Record Array Element	`DBGetVariantArrayValue`
Get Record Array Column	`DBGetVariantArrayColumn`
Free Variant Array	`DBFreeVariantArray`
Close SQL Statement	`DBCloseSQLStatement`
Discard SQL Statement	`DBDiscardSQLStatement`
SQL with Parameters	
Prepare SQL Statement	`DBPrepareSQL`
Refresh Parameters	`DBRefreshParams`
Create Parameters	
Create Integer Parameter	`DBCreateParamInt`
Create Short Integer Parameter	`DBCreateParamShort`
Create Float Parameter	`DBCreateParamFloat`
Create Double Parameter	`DBCreateParamDouble`
Create String Parameter	`DBCreateParamChar`
Create Binary Parameter	`DBCreateParamBinary`
Set Parameters	
Set Parameter to Integer	`DBSetParamInt`
Set Parameter to Short	`DBSetParamShort`
Set Parameter to Float	`DBSetParamFloat`
Set Parameter to Double	`DBSetParamDouble`
Set Parameter to String	`DBSetParamChar`
Set Parameter to Binary	`DBSetParamBinary`
Set Parameter to Variant	`DBSetParamVariant`
Execute Prepared SQL	`DBExecutePreparedSQL`
Set Parameter Attribute	`DBSetParamAttribute`
Get Parameter Attribute	`DBGetParamAttribute`
Get Parameter Values	
Get Parameter as Integer	`DBGetParamInt`
Get Parameter as Short	`DBGetParamShort`
Get Parameter as Float	`DBGetParamFloat`
Get Parameter as Double	`DBGetParamDouble`
Get Parameter as Sting	`DBGetParamChar`

Table 5-1. Function Tree for the SQL Toolkit Library (Continued)

Class/Panel Name	Function Name
Get Parameter as String Buffer	`DBGetParamCharBuffer`
Get Parameter as Binary	`DBGetParamBinary`
Get Parameter as Binary Buf	`DBGetParamBinaryBuffer`
Get Parameter as Variant	`DBGetParamVariant`
Close Prepared SQL	`DBClosePreparedSQL`
Put Values Directly to Record	
Put String in Record	`DBPutColChar`
Put Short Integer in Record	`DBPutColShort`
Put Integer in Record	`DBPutColInt`
Put Float in Record	`DBPutColFloat`
Put Double in Record	`DBPutColDouble`
Put Null in Record	`DBPutColNull`
Put Binary Data in Record	`DBPutColBinary`
Put Variant in Record	`DBPutColVariant`
Get Values Directly from Record	
Get String from Record	`DBGetColChar`
Get String Buffer	`DBGetColCharBuffer`
Get Short from Record	`DBGetColShort`
Get Integer from Record	`DBGetColInt`
Get Float from Record	`DBGetColFloat`
Get Double from Record	`DBGetColDouble`
Get Binary Data from Record	`DBGetColBinary`
Get Binary Buffer	`DBGetColBinaryBuffer`
Get Variant from Record	`DBGetColVariant`
Obsolete or Deprecated	
Allow Previous or Random Fetch	`DBAllowFetchAnyDirection`
Clear Parameter	`DBClearParam`

SQL Toolkit Library Function Reference

The SQL Toolkit Library allows communication with ODBC-compliant databases that you can purchase from a variety of vendors. The library supplies a set of high-level functions for various database interactions. The following function descriptions are in alphabetical order.

DBActivateMap

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

Purpose

Activates a map by constructing a SQL `SELECT` statement based on the map and table name, executing the statement, and binding mapped program variables to the resulting columns.

Parameters

Input

Name	Type	Description
mapHandle	integer	Handle to the map that `DBBeginMap` returns.
tableName	char []	Name of database table to use with the map.

Return Value

Name	Type	Description
statementHandle	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned 0 on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Example

```
hmap = DBBeginMap(hdbc);
resCode = DBMapColumnToChar(hstmt, "ser_num", 11, serialNum,
                  &sNumStatus, "");
resCode = DBMapColumnToDouble(hmap, "measurement",
                    &measurement,
                    &measStatus);
...
hstmt = DBActivateMap(map, "testlog");
while (DBFetchNext(hstmt) == DB_SUCCESS) {
    ...
}
resCode = DBDeactivateMap(hmap);
```

See Also

`DBBeginMap`, `DBDeactivateMap`, `DBMapColumnTo` functions

DBActivateSQL

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

Purpose

Activates a SQL statement. Calling `DBActivateSQL` is equivalent to calling `DBNewSQLStatement` and then `DBOpenSQLStatement`.

Note ***To use SQL parameters you must use*** `DBPrepareSQL` ***and*** `DBExecutePreparedSQL` ***instead of*** `DBActivateSQL`**.**

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` returns.
SQLStatement	char []	SQL statement to activate.

Return Value

Name	Type	Description
statementHandle	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned 0 on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:*** `DBSetBackwardCompatibility(110);`

Example

```
hdbc = DBConnect("DSN=CVI32_Samples");
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
...
resCode = DBDeactivateSQL(hstmt);
resCode = DBDisconnect(hdbc);
```

See Also

DBDeactivateSQL, DBFetchNext, DBFetchPrev, DBFetchRandom, DBBindCol functions, DBColumnName, DBColumnType, DBSetAttributeDefault, DBNewSQLStatement, DBOpenSQLStatement

DBAllowFetchAnyDirection

```
int status = DBAllowFetchAnyDirection (int connectionHandle, int enable);
```

Purpose

Enables or disables fetching `SELECT` statement results in either direction for a database connection.

`DBAllowFetchAnyDirection` is obsolete. Use `DBSetAttributeDefault` with `ATTR_DB_CURSOR_TYPE` instead.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` returns.
enable	integer	`1` = enable fetching in any direction; `0` = disable fetching in any direction.

Return Value

Name	Type	Description
status	integer	Result code that `DBAllowFetchAnyDirection` returns. This function returns the set of result codes listed in the function description for `DBError`.

Example

```
resCode = DBAllowFetchAnyDirection(hdbc,TRUE);
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
numRecs = DBNumberOfRecords(hstmt)
/* Fetch the last record. */
resCode = DBFetchRandom(hstmt,numRecs);
...
resCode = DBDeactivateSQL();
```

See Also

`DBFetchNext`, `DBFetchPrev`, `DBFetchRandom`

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 you fetch a record. You can also use the map with `DBCreateTableFromMap` to create a new database table.

Parameter

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` returns.

Return Value

Name	Type	Description
mapHandle	integer	Handle to the new map. A value less than or equal to `0` indicates an error. This value is a parameter to `DBCreateTableFromMap`, `DBActivateMap`, and the `DBMapColumnTo` functions.

☞ **Note** ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned 0 on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:*** `DBSetBackwardCompatibility(110);`

Example

```
hmap = DBBeginMap(hdbc);
resCode = DBMapColumnToChar(hstmt, "SER_NUM", 11, serialNum,
                  &sNumStatus, "");
resCode = DBMapColumnToDouble(map, "MEAS1",
                    &measurement, &measStatus);
...
hstmt = DBActivateMap(hmap, "TESTLOG");
while (DBFetchNext(hstmt) == 0) {
    ...
}
resCode = DBDeactivateMap(hmap);
```

See Also

DBMapColumnTo functions, DBCreateTableFromMap, DBActivateMap, DBFetchNext, DBFetchPrev, DBFetchRandom, DBDeactivateMap

DBBeginTran

```
int status = DBBeginTran (int connectionHandle);
```

Purpose

Starts a transaction on a database connection. After a transaction begins, the SQL `INSERT`, `UPDATE`, and `DELETE` statements that you execute, as well as changes you make with `DBCreateRecord`, `DBDeleteRecord`, and `DBPutRecord` are not committed to the database until you call `DBCommit`. The following table explains the functions for database transactions.

Table 5-2. Functions for Database Transactions

Function	Purpose	Discussion
`DBBeginTran`	Begins a transaction with a database.	Only one call to `DBBeginTran` can be active on a connection at any time. You can make subsequent calls to `DBBeginTran` after you call either `DBCommit` or `DBRollback`.
`DBCommit`	Saves changes, frees all database locks for a transaction, and ends transaction.	Only works during a `DBBeginTran` transaction.
`DBRollback`	Discards changes, frees all database locks for a transaction, and ends transaction.	Only works during a `DBBeginTran` transaction.

Note ***Keep in mind the following restrictions regarding transactions:***

- ***Some database systems do not support transactions.***
- ***If you execute an*** `INSERT`, `UPDATE`, ***or*** `DELETE` ***statement without first calling*** `DBBeginTran`, ***the toolkit automatically commits database changes and releases all database locks.***
- ***During a transaction, you must precede a call to*** `DBDisconnect` ***with a call to*** `DBCommit` ***or*** `DBRollback`.
- ***Some database systems do not allow you to begin a transaction after you have executed a SQL statement with*** `DBActivateSQL`, `DBImmediateSQL`, `DBActivateMap`, `DBOpenSQLStatement`, ***or*** `DBExecutePreparedSQL`.

Parameter

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` returns.

Return Value

Name	Type	Description
status	integer	Result code that `DBBeginTran` returns. `DBBeginTran` returns the set of result codes listed in the function description for `DBError`.

Example

```
hdbc = DBConnect("DSN=MSSS;UID=shawkins;SRVR=PENNY");
...
resCode = DBBeginTran(hdbc);
hstmt = DBActivateSQL(hdbc,
    "UPDATE EMP SET SALARY = SALARY * 1.1");
resCode = DBDeactivate(hstmt);
resCode = DBCommit(hdbc);
resCode = DBDisconnect(hdbc);
```

See Also

`DBCommit`, `DBRollback`

DBBindColBinary

```
int status = DBBindColBinary (int statementHandle, int columnNumber, unsigned
                              long maximumLength, char locationforValue[], long
                              *locationforStatus);
```

Purpose

Specifies the value and status variables in your program that receive the value and length of a column when you fetch a record. You do not have to bind all columns in the statement. You can bind columns in any order.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL` or any of the functions that return a statement handle.
columnNumber	integer	Column number to bind to the specified variables. The first column number is one.
maximumLength	unsigned long integer	Size of the value variable in bytes.
locationforValue	any array	Pointer to the variable that receives the binary value for the column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record.

Return Value

Name	Type	Description
status	integer	Result code that `DBBindColBinary` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values:

Value Name	Value	Description
DB_TRUNCATION	-1	Data is truncated.
DB_NULL_DATA	-2	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
unsigned char *toDBBits = NULL;
int bitsStatus = 0;
int bitsSize = 6;
...
hstmt = DBActivateSQL(hdbc, "SELECT THE BITS FROM BINTEST");

fromDBBits = malloc(bitsSize);
dbStatus = DBBindColBinary(hstmt, 1, bitsSize, fromDBBits,
                 &bitsStatus);

while ((dbStatus = DBFetchNext(hstmt) == DB_SUCCESS) {
    /* Use the value. */
}

dbStatus = DBDeactivateSQL(hstmt);
free(fromDBBits);
hstmt = 0;
```

See Also

DBFetchNext, DBFetchPrev, DBFetchRandom, DBActivateSQL, DBPutRecord, DBDeactivateSQL

DBBindColChar

```
int status = DBBindColChar (int statementHandle, int columnNumber, unsigned
                            long maximumLength, char locationforValue[], long
                            *locationforStatus, char formatString[]);
```

Purpose

Specifies the value and status variables in your program that are to receive the value and status of a column when you fetch a record. You do not have to bind all columns in the statement. You can bind columns in any order.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL` or any of the functions that return a statement handle.
columnNumber	integer	Column number to bind to the specified variables. The first column number is `1`.
maximumLength	unsigned long integer	Size of the value variable in bytes. The toolkit uses one byte of the variable for the string termination character, NUL.
locationforValue	char []	Pointer to the variable that receives the null-terminated character string value for the column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record.
formatString	char []	Format string. Use the empty string, `""`, if you want the default format. See Appendix C, *Format Strings*, for details about formatting.

Return Value

Name	Type	Description
status	integer	Result code that `DBBindColChar` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values for **locationforStatus**:

Value Name	Value	Description
`DB_TRUNCATION`	`-1`	Data is truncated.
`DB_NULL_DATA`	`-2`	Null data.
(none)	positive integer	Number of bytes fetched.

☞ **Note** ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
char serialNum[11];
long serialNumLen;
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
serialNumLen = 11;
DBBindColChar(hstmt, 1, serialNumLen, serialNum, &serialNumStat, "");
/* More variable bindings. */
...
while (DBFetchNext(hstmt) == 0) {
    if (serialNumStat == DB_NULLDATA)
       ...
    if (serialNumStat == DB_TRUNCATION)
       ...
    printf("Serial Number: %s\n",serialNum);
    ...
}
resCode = DBDeactivateSQL();
```

See Also

DBFetchNext, DBFetchPrev, DBFetchRandom, DBActivateSQL, DBPutRecord, DBDeactivateSQL

DBBindColDouble

```
int status = DBBindColDouble (int statementHandle, int columnNumber, double
                              *locationforValue, long *locationforStatus);
```

Purpose

Specifies the value and status variables in your program that are to receive the value and status of a column when you fetch a record. The `DBFetch` function converts the data to a double-precision floating-point value. You do not have to bind all columns in the statement. You can bind columns in any order.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL` or any of the functions that return a statement handle.
columnNumber	integer	Column number to bind to the specified variables. The first column number is `1`.
locationforValue	double-precision (passed by reference)	Pointer to the double-precision variable that receives the value of a column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record.

Return Value

Name	Type	Description
status	integer	Result code that `DBBindColDouble` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values:

Value Name	Value	Description
DB_TRUNCATION	-1	Data is truncated.
DB_NULL_DATA	-2	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** DB_NULL_DATA ***to place a NULL value into a column as follows: set the status value for that column to*** DB_NULL_DATA***, then call*** DBPutRecord***. To prevent*** DBPutRecord ***from updating a column, set the status value to*** DB_NO_DATA_CHANGE***.*** DB_NO_DATA_CHANGE ***is useful when the record contains read-only columns.***

Example

```
double measurement;
long   measStat;
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
/* Other variable bindings. */
...
DBBindColDouble(hstmt, 6, &measurement, &measStat);
/* More variable bindings. */
...
while (DBFetchNext(hstmt) == 0) {
    ...
    if (measStat == DB_NULL_DATA)
       ...
    printf("Measurement: %f\n",measurement);
    ...
}
resCode = DBDeactivateSQL();
```

See Also

DBFetchNext, DBFetchPrev, DBFetchRandom, DBActivateSQL, DBDeactivate functions, DBPutRecord

DBBindColFloat

```
int status = DBBindColFloat (int statementHandle, int columnNumber, float
                             *locationforValue, long *locationforStatus);
```

Purpose

Specifies the value and status variables in your program that are to receive the value and length of a column when you fetch a record. The `DBFetch` function converts the data to a single-precision floating point value. You do not have to bind all columns in the statement. You can bind columns in any order.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL` or any of the functions that return a statement handle.
columnNumber	integer	Column number to bind to the specified variables. The first column number is `1`.
locationforValue	float (passed by reference)	Pointer to the floating-point variable that receives the value of a column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record.

Return Value

Name	Type	Description
status	integer	Result code that `DBBindColFloat` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values:

Value Name	Value	Description
DB_TRUNCATION	-1	Data is truncated.
DB_NULL_DATA	-2	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
float measurement;
long measStat;
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
/* Other variable bindings. */
...
DBBindColFloat(hstmt, 6, &measurement, &measStat);
/* More variable bindings. */
...
while (DBFetchNext(hstmt) == 0) {
    ...
    if (measStat == DB_NULL_DATA)
    ...
    printf("Measurement: %f\n",measurement);
    ...
}
resCode = DBDeactivateSQL();
```

See Also

DBFetchNext, DBFetchPrev, DBFetchRandom, DBActivateSQL, DBPutRecord, DBDeactivateSQL

DBBindColInt

```
int status = DBBindColInt (int statementHandle, int columnNumber, int
                           *locationforValue, long *locationforStatus);
```

Purpose

Specifies the value and length variables in your program that are to receive the value and length of a column when you fetch a record. The `DBFetch` function converts the data to a 4-byte integer (a `long int` or `int` in LabWindows/CVI). You do not have to bind all columns in the statement. You can bind columns in any order.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL` or any of the functions that return a statement handle.
columnNumber	integer	Column number to bind to the specified variables. The first column number is `1`.
locationforValue	integer (passed by reference)	Pointer to the integer that receives the value of a column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record.

Return Value

Name	Type	Description
status	integer	Result code that `DBBindColInt` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values:

Value Name	Value	Description
DB_TRUNCATION	-1	Data is truncated.
DB_NULL_DATA	-2	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
long numTries;
long numTriesStat;
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
/* Other variable bindings. */
...
DBBindColInt(hstmt, 5, &numTries, &numTriesStat);
/* More variable bindings. */
...
while (DBFetchNext(hstmt) == 0) {
    ...
    if (numTriesStat == DB_NULL_DATA)
       ...
    if (numTriesStat == qeTRUNCATION)
       ...
    printf("Number of tries: %ld\n",numTries);
    ...
}
resCode = DBDeactivateSQL();
```

See Also

DBFetchNext, DBFetchPrev, DBFetchRandom, DBActivateSQL, DBDeactivateSQL, DBPutRecord

DBBindColShort

```
int status = DBBindColShort (int statementHandle, int columnNumber, short
                            *locationforValue, long *locationforStatus);
```

Purpose

Specifies the value and status variables in your program that are to receive the value and length of a column when you fetch a record. The `DBFetch` function converts the data to a 2-byte short integer. You do not have to bind all columns in the statement. You can bind columns in any order.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL` or any of the functions that return a statement handle.
columnNumber	integer	Column number to bind to the specified variables. The first column number is `1`.
locationforValue	short integer (passed by reference)	Pointer to the short integer that receives the value of a column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record.

Return Value

Name	Type	Description
status	integer	Result code that `DBBindColShort` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values:

Value Name	Value	Description
DB_TRUNCATION	-1	Data is truncated.
DB_NULL_DATA	-2	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
short numTries;
long numTriesStat;
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
/* Other variable bindings. */
...
DBBindColShort(hstmt, 5, &numTries, &numTriesStat, "");
/* More variable bindings. */
...
while (DBFetchNext(hstmt) == 0) {
    ...
    if (numTriesStat == DB_NULL_DATA)
       ...
    if (numTriesStat == DB_TRUNCATION)
       ...
    printf("Number of tries: %d\n",numTries);
    ...
}
resCode = DBDeactivateSQL();
```

See Also

DBFetchNext, DBFetchPrev, DBFetchRandom, DBActivateSQL, DBDeactivateSQL, DBPutRecord

DBCancelRecordChanges

```
int status = DBCancelRecordChanges (int statementHandle);
```

Purpose

Cancels pending changes to the current record or discards a newly added record.

If you are adding a new record when you call `DBCancelRecordChanges`, the record that was current before the `DBCreateRecord` call becomes the current record again.

If you have not changed the current record or you have not added a new record, a call to `DBCancelRecordChanges` generates an error.

Some ODBC drivers and ADO providers do not allow you to deactivate a SQL statement if there is a record that has pending changes. For example, some drivers and providers will not allow you to deactivate a SQL statement if an error occurs during the preparation of a new record. You can use `DBCancelRecordChanges` to cancel the pending changes so your program can deactivate the statement.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL` or any of the functions that return a statement handle.

Return Value

Name	Type	Description
status	integer	Result code that `DBCancelRecordChanges` returns. Result codes are the same as those that `DBError` returns.

Example

```
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
...
resCode = DBCreateRecord(hstmt, 1);
...
resCode = DBPutColInt(hstmt, 2, i);
if (resCode < 0) {
    DBCancelRecordChanges(hstmt);
    goto DB_Error;
}
...
resCode = DBPutRecord(hstmt);
...
resCode = DBDeactivateSQL();
```

See Also

DBActivateSQL, DBActivateMap, DBCreateRecord, DBBindCol functions, DBMapColumnTo functions

DBClearParam

```
int status = DBClearParam (int statementHandle, int index);
```

Purpose

Clears the value of a parameter by setting the value to be empty. You do not have to call this function before changing the value of a parameter. You do not have to call this function before closing a prepared statement.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle from `DBActivateSQL`.
index	integer	Index of the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBClearParam` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

DBCloseConnection

```
int status = DBCloseConnection (int connectionHandle);
```

Purpose

Closes the specified connection. The connection continues to exist and you can reopen it with `DBOpenConnection` until you discard it with `DBDiscardConnection`.

Parameter

Input

Name	Type	Description
connectionHandle	integer	Connection handle from `DBConnect` or `DBNewConnection`.

Return Value

Name	Type	Description
status	integer	Result code that `DBCloseConnection` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hdbc = DBNewConnection();
resCode = DBSetConnectionAttribute(hdbc,
                              ATTR_DB_CONN_CONNECTION_TIMEOUT, 100);
resCode = DBSetConnectionAttribute(hdbc,
        ATTR_DB_CONN_ISOLATION_LEVEL, DB_ISOLATION_LEVEL_ISOLATED);
resCode = DBOpenConnection(hdbc);
...
resCode = DBCloseConnection(hdbc);
resCode = DBDiscardConnection(hdbc);
```

See Also

`DBDisconnect`, `DBNewConnection`, `DBOpenConnection`, `DBSetConnectionAttribute`, `DBGetConnectionAttribute`, `DBDiscardConnection`

DBClosePreparedSQL

```
int status = DBClosePreparedSQL (int statementHandle);
```

Purpose

Closes a statement that you execute with `DBExecutePreparedSQL`. You must close the statement before you can examine output parameters of a stored procedure. You must discard a closed statement with `DBDiscardSQLStatement` to properly free resources. You do not have to call `DBClosePreparedSQL` for stored procedures that use only input parameters; you can call `DBDeactivateSQL` to close and discard the statement in a single step.

☞ **Note** `DBClosePreparedSQL` ***discards the variable bindings for the statement. If you execute the statement again, you must create new variable bindings.***

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns.

Return Value

Name	Type	Description
status	integer	Result code that `DBClosePreparedSQL` returns. `DBClosePreparedSQL` returns the set of result codes listed in the function description for `DBError`.

Example

```
/* Create a stored procedure.              */
/* This syntax is for Microsoft SQL Server. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam int, @OutParam int OUTPUT ) as  \
        select @OutParam = @InParam + 10 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam +10");

/* Prepare a statement that calls the stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_UNKNOWN);
```

```
/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamShort(hstmt, 1, 10);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* process records returned by the stored procedure. */
}

/* Close the statement.  Output parameters are invalid.  */
/* until you close the statement.                        */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamShort(hstmt, 0, &retParam);
resCode = DBGetParamShort(hstmt, 1, &inParam);
resCode = DBGetParamShort(hstmt, 2, &outParam);

/* Discard the statement. */
hstmt = DBDiscardSQLStatement(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL

DBCloseSQLStatement

```
int status = DBCloseSQLStatement (int statementHandle);
```

Purpose

Closes a statement that `DBOpenSQLStatement` or `DBActivateSQL` opened. The statement continues to exist and can be reopened with `DBOpenSQLStatement` until you discard it with `DBDiscardSQLStatement`. Calling `DBCloseSQLStatement` and then `DBDiscardSQLStatement` is equivalent to calling `DBDeactivateSQL`.

Note `DBCloseSQLStatement` ***discards the variable bindings for the statement. If you reopen the statement, you must create new variable bindings.***

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBNewSQLStatement` or `DBActivateSQL` returns.

Return Value

Name	Type	Description
status	integer	Result code that `DBCloseSQLStatement` returns. `DBCloseSQLStatement` returns the set of result codes listed in the function description for `DBError`.

Example

```
hstmt = DBNewSQLStatement (hdbc, "SELECT UUT_NUM, MEAS1,\
                        MEAS2 FROM TESTRES");
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_MAX_RECORDS,
          1);
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_CACHE_SIZE,
          10);
resCode = DBOpenSQLStatement(hstmt);
...
resCode = DBGetStatementAttribute(hstmt, ATTR_DB_STMT_RECORD_COUNT,
                                  &recordCount
...
resCode = DBCloseSQLStatement(hstmt);
resCode = DBDiscardSQLStatement(hstmt);
```

See Also

DBNewSQLStatement, DBOpenSQLStatement, DBDiscardSQLStatement, DBSetStatementAttribute, DBGetStatementAttribute

DBColumnName

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

Purpose

Returns the name of a column.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL`, `DBActivateMap`, or any of the functions that return a statement handle.
columnNumber	integer	Column number for which the name is to be returned. The first column number is `1`.

Return Value

Name	Type	Description
columnName	char *	Pointer to the returned column name. The string is stored in a buffer that the toolkit maintains. You must copy the string out of this buffer before you call another toolkit function, because the next function might use the same buffer. The column name is empty string, `""`, if the column is an expression in the SQL `SELECT` statement. The column name is `NULL` if an error occured.

Example

```
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
colName = DBColumnName(hstmt, 1);
...
resCode = DBDeactivate(hstmt);
```

See Also

`DBActivateSQL`, `DBActivateMap`

DBColumnType

```
int columnType = DBColumnType (int statementHandle, int columnNumber);
```

Purpose

Returns the data type for a column in a SQL `SELECT` statement.

Note ***Prior to version 2.0, the SQL Toolkit returned only*** `DB_CHAR`, `DB_VARCHAR`, `DB_DECIMAL`, `DB_INTEGER`, `DB_SMALLINT`, `DB_FLOAT`, `DB_DOUBLEPRECISION`, `DB_DATETIME` ***types. To minimize changes to programs which depend on this behavior, make the following function call to set the compatibility mode to version 1.1:*** `DBSetBackwardCompatibility(110);`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL`, `DBActivateMap`, or any of the functions that return a statement handle.
columnNumber	integer	Column number for which the type is to be returned. The first column number is `1`.

Return Value

Name	Type	Description
columnType	integer	Returned data type. The type is the LabWindows/CVI SQL Toolkit common type which might not be the same as the type in the underlying database.

Return Codes

Constant	Description
`DB_TINYINT`, `DB_SMALLINT`, `DB_INTEGER`, `DB_BIGINT`	1-, 2-, 4-, and 8-byte integers, respectively.
`DB_UNSIGNEDTINYINT`, `DB_UNSIGNEDSMALLINT`, `DB_UNSIGNEDINT`, `DB_UNSIGNEDBIGINT`	1-, 2-, 4-, and 8-byte unsigned integers, respectively.
`DB_FLOAT`	4-byte floating point

Constant	Description
DB_DOUBLEPRECISION	8-byte double precision.
DB_CURRENCY	Fixed-point number with four digits to the right of the decimal point. Stored in a 4-byte integer scaled by 10000.
DB_DECIMAL	Exact numeric value with a fixed precision and scale.
DB_NUMERIC	Exact numeric value with a fixed precision and scale.
DB_BOOLEAN	Boolean value.
DB_ERROR	4-byte error code.
DB_USERDEFINED	User defined.
DB_GUID	Globally unique identifier.
DB_DATE	days_since_1899_12_30.fraction_of_day, stored as a double-precision value.
DB_DBDATE	yyyymmdd.
DB_TIME	hhmmss.
DB_DATETIME	yyyymmddhhmmss plus a fraction in billionths.
DB_BSTR	null-terminated character string.
DB_CHAR	Fixed-length string.
DB_VARCHAR	String value (parameter only).
DB_LONGVARCHAR	Long string value (parameter only).
DB_WCHAR	Wide character null-terminated string.
DB_VARWCHAR	Wide character null-terminated string (parameter only).
DB_LONGVARWCHAR	Long null-terminated string (parameter only).
DB_BINARY	Binary value.
DB_VARBINARY	Variable-length binary value.
DB_LONGVARBINARY	Long binary value.

Example

```
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
dataType = DBColumnType(hstmt, 1); ...
resCode = DBDeactivateSQL(hstmt);
```

See Also

DBActivateSQL, DBActivateMap

DBColumnWidth

```
int columnWidth = DBColumnWidth (int statementHandle, int columnNumber);
```

Purpose

Returns the width of a column. The width is the size, in bytes, of the longest value that a column can store.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL`, `DBActivateMap`, or any of the functions that return a statement handle.
columnNumber	integer	Column number for which the width is to be returned. The first column number is `1`.

Return Value

Name	Type	Description
columnWidth	integer	Width of the column in bytes.

Example

```
hdbc = DBConnect("DSN=CVI32_Samples");
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
colWidth = DBColumnWidth(hstmt, 1);
...
resCode = DBDeactivate(hstmt);
resCode = DBDisconnect(hdbc);
```

See Also

`DBActivateSQL`, `DBActivateMap`

DBCommit

```
int status = DBCommit (int connectionHandle);
```

Purpose

Commits all changes that you make using the SQL statements `INSERT`, `UPDATE`, or `DELETE` during a call to `DBBeginTran`. You must call `DBBeginTran` to begin a transaction before you can call `DBCommit` to save all changes.

Parameter

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returns.

Return Value

Name	Type	Description
status	integer	Result code that `DBCommit` returns. `DBCommit` returns the set of result codes listed in the function description for `DBError`.

Example

```
hdbc = DBConnect("DSN=QESS;UID=shawkins;SRVR=PENNY");
...
resCode = DBBeginTran(hdbc);
hstmt = DBActivateSQL(hdbc,
    "UPDATE EMP SET SALARY = SALARY * 1.1");
resCode = DBDeactivateSQL(hstmt);
resCode = DBCommit(hdbc);
resCode = DBDisconnect(hdbc);
```

See Also

`DBBeginTran`, `DBRollback`

DBConnect

```
int connectionHandle = DBConnect (char connectionString[]);
```

Purpose

Opens a connection to a database system to allow execution SQL statements. Calling `DBConnect` is equivalent to calling `DBNewConnection` and then `DBOpenConnection`.

Parameter

Input

Name	Type	Description
connectionString	char []	Specifies a data source as a detailed connection string containing a series of argument = value statements separated by semicolons.

Return Value

Name	Type	Description
connectionHandle	integer	Database connection handle that identifies the connection and is a parameter to other functions. If the handle is less than or equal to `0`, the program could not open the connection.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Note ***You must use*** `DBInit` ***before*** `DBConnect` ***when your program is multithreaded.***

Parameter Discussion

The following attributes are supported at the Active Data Object (ADO) level. Any other arguments are passed directly to the provider without any processing by ADO.

Attribute	Description
Provider	Name of ADO provider to use for the connection. If not specified the connection uses default ODBC provider.
Data Source	Name of the data source to use for the connection, for example an Microsoft Access database registered as an ODBC data source.
User ID	User name to use when opening the connection.
Password	Password to use when opening the connection.
File Name	Provider specific file that contains preset connection information.

Most ODBC data sources support the following attributes. See the driver documentation for each database for additional attributes.

Attribute	Description
DSN	Name of the ODBC data source to use for the connection. In many cases, DSN is the only parameter you must specify.
DLG	When enabled (`DLG=1`) displays a dialog box that allows user input of connection string information.
UID	User ID or name.
PWD	Password.
MODIFYSQL	Set to `1`, support ODBC compliant SQL. Set to `0`, support native SQL of the underlying database.

Examples

Use the following call to connect to CVI Sample tables:

```
hdbc = DBConnect("DSN=CVI32_Samples");
...
res_code = DBDisconnect();
```

See Also

`DBDisconnect`, `DBNewConnection`, `DBOpenConnection`

DBCreateParamBinary

```
int status = DBCreateParamBinary (int statementHandle, char *parameterName,
                                  tDBParameterDirection parameterDirection, void
                                  *initialValue, int valueSizeinBytes);
```

Purpose

Creates a binary parameter for a stored procedure or SQL statement that you previously prepared with `DBPrepareSQL`. You cannot create parameters for statements that you execute with `DBActivateSQL` or `DBOpenSQLStatement`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL` or `DBNewSQLStatement`.
parameterName	char *	The name of the parameter. Use the empty string `""` to create an unnamed parameter.
parameterDirection	`tDBParameterDirection`	Direction of the parameter: input, output, input/output, or return value.
initialValue	void pointer	Binary initial value for the parameter.
valueSizeinBytes	integer	Size of the value in bytes.

Return Value

Name	Type	Description
status	integer	Result code that `DBCreateParamBinary` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Input Parameter Example

```
unsigned char data[10];

/* Note that binary data allows embedded NUL characters. */
data[0]='N';data[1]=0;data[2]='C';data[3]=0;
data[4]='B';data[5]='I';data[6]='S';data[7]='O';
data[8]='P';data[9]=0;
...
hstmt = DBPrepareSQL (hdbc, "SELECT NAME, DRESS_SIZE,\
        FROM DEVGUYS WHERE NAME = ?");
resCode = DBCreateParamChar(hstmt, "", DB_PARAM_INPUT, data, 10);
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
hstmt = DBDeactivateSQL(hstmt);
```

Output Parameter Example

```
/* This example works with Microsoft SQL Server. */
unsigned char inParam[6] = "in";
unsigned char readInParam[6];
unsigned char *outParam;
char* retParam;
...
/* Create the stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest( \
          @InParam binary(6), @OutParam binary(6) OUTPUT ) as \
          select @OutParam = @InParam SELECT * FROM Authors \
          WHERE State <> 'CA' return 10.1");

/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                        DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_CVItest");
/* Set the command type attribute back to the default */
/* for future commands.                               */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                        DB_COMMAND_UNKNOWN);

/* Put some data in the input variable           */
/* note that using binary allows embedded nulls. */
inParam[0]='N';inParam[1]='C';inParam[2]='B';
inParam[3]='\0'; inParam[4]='O';inParam[5]='K';
outParam = malloc(6);
```

```
/* Create the parameters. */
resCode = DBCreateParamChar(hstmt, "", DB_PARAM_RETURN_VALUE,
                            retParam, 6);
resCode = DBCreateParamBinary(hstmt, "InParam", DB_PARAM_INPUT, "",
                              6);
resCode = DBCreateParamBinary(hstmt, "OutParam", DB_PARAM_OUTPUT, "",
                              6);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* process records returned by the stored procedure. */
}

/* Close the statement.  Output values are invalid  */
/* until you close the statement.                   */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamChar(hstmt, 0, &retParam, "");
resCode = DBGetParamBinaryBuffer(hstmt, 1, readInParam, 6);
resCode = DBGetParamBinary(hstmt, 2, (void **)&(outParam));
DBFree(retParam);
DBFree(outParam);

/* Deactivate the statement. */
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamBinary, DBGetParamBinary, DBGetParamBinaryBuffer, DBClosePreparedSQL

DBCreateParamChar

```
int status = DBCreateParamChar (int statementHandle, char *parameterName,
                                tDBParameterDirection parameterDirection, char
                                *initialValue, int valueSizeinBytes);
```

Purpose

Creates a string parameter for a stored procedure or SQL statement that you previously prepared with `DBPrepareSQL`. You cannot create parameters for statements that you execute with `DBActivateSQL` or `DBOpenSQLStatement`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL` or `DBNewSQLStatement`.
parameterName	char *	The name of the parameter. Use the empty string `""` to create an unnamed parameter.
parameterDirection	`tDBParameterDirection`	Direction of the parameter: input, output, input/output, or return value.
initialValue	char *	String value for the parameter.
valueSizeinBytes	integer	Size of the value in bytes.

Return Value

Name	Type	Description
status	integer	Result code that `DBCreateParamChar` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Input Parameter Example

```
char uut[10] = "yyd2860b1";
...
hstmt = DBPrepareSQL (hdbc, "SELECT UUT_NUM, MEAS1,\
        MEAS2 FROM TESTRES WHERE UUT_NUM = ?");
resCode = DBCreateParamChar(hstmt, "",
                   DB_PARAM_INPUT,"YYD2860b1", 10);
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
hstmt = DBDeactivateSQL(hstmt);
```

Output Parameter Example

```
/* This example works with Microsoft SQL Server. */
char inParam[11] = "in";
char readInParam[11];
char outParam[11];
char* retParam;
...
/* Create the stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest( \
       @InParam char(10), @OutParam char(10) OUTPUT ) as \
       select @OutParam = 'out' SELECT * FROM Authors \
       WHERE State <> 'CA' return 10.1");

/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_CVItest");

/* Set the command type attribute back to the default  */
/* for future commands.                                */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_UNKNOWN);

/* Create the parameters */
resCode = DBCreateParamChar(hstmt, "",
                  DB_PARAM_RETURN_VALUE,
                  inParam, 10);
resCode = DBCreateParamChar(hstmt, "InParam",
                  DB_PARAM_INPUT, "", 10);
```

```
resCode = DBCreateParamChar(hstmt, "OutParam",
                  DB_PARAM_OUTPUT, "", 10);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by the stored procedure. */
}

/* Close the statement.  Output values are invalid  */
/* until you close the statement.                   */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamChar(hstmt, 0, &retParam, "");
resCode = DBGetParamCharBuffer(hstmt, 1, readInParam, 10, "");
resCode = DBGetParamCharBuffer(hstmt, 2, outParam, 10, "");
printf("return param %s input param %s output param %s\n",
       retParam, inParam, outParam);
free(retParam);

/* Deactivate the statement. */
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamChar, DBGetParamChar, DBGetParamCharBuffer, DBClosePreparedSQL

DBCreateParamDouble

```
int status = DBCreateParamDouble (int statementHandle, char *parameterName,
                                  tDBParameterDirection parameterDirection, double
                                  initialValue);
```

Purpose

Creates a double-precision parameter for a stored procedure or SQL statement that you previously prepared with `DBPrepareSQL`. You cannot create parameters for statements that you create with `DBActivateSQL` or `DBNewSQLStatement`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL` or `DBNewSQLStatement`.
parameterName	char *	Name of the parameter. Use the empty string `""` to create an unnamed parameter.
parameterDirection	`tDBParameterDirection`	Direction of the parameter: input, output, input/output, or return value.
value	double-precision	Initial value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBCreateParamDouble` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Input Parameter Example

```
hstmt = DBPrepareSQL (hdbc, "SELECT UUT_NUM, MEAS1,\
        MEAS2 FROM TESTRES WHERE MEAS1 > ?");
resCode = DBCreateParamDouble(hstmt, "",
                  DB_PARAM_INPUT, 1.5);
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
hstmt = DBDeactivateSQL(hstmt);
```

Output Parameter Example

```
/* This example works with Microsoft SQL Server. */
/* Create a stored procedure with input, output  */
/* and return value parameters.                  */
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest( \
        @InParam float, @OutParam float OUTPUT ) as select \
        @OutParam =  @InParam + 10.1 SELECT * FROM Authors \
        WHERE State <> 'CA' return @OutParam + 10.1");

/* Set command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_STORED_PROC);
/* Prepare the SQL statement. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
/* Set the command type attribute back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_UNKNOWN);

/* Create the three parameters. */
resCode = DBCreateParamDouble(hstmt, "",
                  DB_PARAM_RETURN_VALUE, -1);
resCode = DBCreateParamDouble(hstmt, "InParam",
                  DB_PARAM_INPUT, 10.1);
resCode = DBCreateParamDouble(hstmt, "OutParam",
                  DB_PARAM_OUTPUT, -1);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by stored procedure. */
}

/* Close the statement.  The output parameters are */
/* invalid until you close the statement.          */
resCode = DBClosePreparedSQL(hstmt);
```

```
/* Examine the parameters */
resCode = DBGetParamDouble(hstmt, 0, &retParam);
resCode = DBGetParamDouble(hstmt, 1, &inParam);
resCode = DBGetParamDouble(hstmt, 2, &outParam);

hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamDouble, DBGetParamDouble, DBClosePreparedSQL

DBCreateParamFloat

```
int status = DBCreateParamFloat (int statementHandle, char *parameterName,
                                 tDBParameterDirection parameterDirection, float
                                 initialValue);
```

Purpose

Creates a floating-point parameter for a stored procedure or SQL statement that you previously prepared with `DBPrepareSQL`. You cannot create parameters for statements that you create with `DBActivateSQL` or `DBNewSQLStatement`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL` or `DBNewSQLStatement`.
parameterName	char *	Name of the parameter. Use the empty string `""` to create an unnamed parameter.
parameterDirection	`tDBParameterDirection`	Direction of the parameter: input, output, input/output, or return value.
initialValue	float	Floating-point initial value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBCreateParamFloat` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Input Parameter Example

```
hstmt = DBPrepareSQL (hdbc, "SELECT UUT_NUM, MEAS1,\
        MEAS2 FROM TESTRES WHERE MEAS1 > ?");
resCode = DBCreateParamFloat(hstmt, "", DB_PARAM_INPUT, 1.5);
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
```

```
    ...
}
hstmt = DBDeactivateSQL(hstmt);
```

Output Parameter Example

```
/* This example works with Microsoft SQL Server.  */
/* Create a stored procedure with input, output  */
/* and return value parameters.                  */
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest( \
        @InParam float, @OutParam float OUTPUT ) as select \
        @OutParam =  @InParam + 10.1 SELECT * FROM Authors \
        WHERE State <> 'CA' return @OutParam + 10.1");

/* Set command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_STORED_PROC);
/* Prepare the SQL statement. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
/* Set the command type attribute back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_UNKNOWN);

/* Create the three parameters. */
resCode = DBCreateParamFloat(hstmt, "", DB_PARAM_RETURN_VALUE, -1);
resCode = DBCreateParamFloat(hstmt, "InParam", DB_PARAM_INPUT, 10.1);
resCode = DBCreateParamFloat(hstmt, "OutParam", DB_PARAM_OUTPUT, -1);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* process records returned by stored procedure. */
}

/* Close the statement.  The output parameters are */
/* invalid until you close the statement.          */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameters. */
resCode = DBGetParamFloat(hstmt, 0, &retParam);
resCode = DBGetParamFloat(hstmt, 1, &inParam);
resCode = DBGetParamFloat(hstmt, 2, &outParam);

hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamFloat, DBGetParamFloat, DBClosePreparedSQL

DBCreateParamInt

```
int status = DBCreateParamInt (int statementHandle, char *parameterName,
                               tDBParameterDirection parameterDirection, int
                               initialValue);
```

Purpose

Creates an integer parameter for a stored procedure or SQL statement that you previously prepared with `DBPrepareSQL`. You cannot create parameters for statements that you create with `DBActivateSQL` or `DBNewSQLStatement`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL` or `DBNewSQLStatement`.
parameterName	char *	Name of the parameter. Use the empty string `""` to create an unnamed parameter.
parameterDirection	`tDBParameterDirection`	Direction of the parameter: input, output, input/output, or return value.
initialValue	integer	Integer initial value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBCreateParamInt` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Input Parameter Example

```
hstmt = DBPrepareSQL (hdbc, "SELECT UUT_NUM, LOOPNUM MEAS1,\
        MEAS2 FROM REC1000 WHERE LOOPNUM > ?");
resCode = DBCreateParamInt(hstmt, "", DB_PARAM_INPUT, 100);
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
hstmt = DBDeactivateSQL(hstmt);
```

Output Parameter Example

```
/* This example works with Microsoft SQL Server.  */
/* Create a stored procedure with input, output  */
/* and return value parameters.                  */
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest( \
        @InParam int, @OutParam int OUTPUT ) as select \
        @OutParam =  @InParam + 10 SELECT * FROM Authors \
        WHERE State <> 'CA' return @OutParam +10");

/* Set command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_STORED_PROC);
/* Prepare the SQL statement. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
/* Set the command type attribute back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_UNKNOWN);

/* Create the three parameters. */
resCode = DBCreateParamInt(hstmt, "",
                 DB_PARAM_RETURN_VALUE, -1);
resCode = DBCreateParamInt(hstmt, "InParam",
                 DB_PARAM_INPUT, 10);
resCode = DBCreateParamInt(hstmt, "OutParam",
                 DB_PARAM_OUTPUT, -1);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by stored procedure. */
}

/* Close the statement.  The output parameters are */
/* invalid until you close the statement.          */
resCode = DBClosePreparedSQL(hstmt);
```

```
/* Examine the parameters. */
resCode = DBGetParamInt(hstmt, 0, &retParam);
resCode = DBGetParamInt(hstmt, 1, &inParam);
resCode = DBGetParamInt(hstmt, 2, &outParam);

hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamInt, DBGetParamInt, DBClosePreparedSQL

DBCreateParamShort

```
int status = DBCreateParamShort (int statementHandle, char *parameterName,
                                 tDBParameterDirection parameterDirection, short
                                 initialValue);
```

Purpose

Creates a parameter for a stored procedure or SQL statement that you previously prepared with `DBPrepareSQL`. You cannot create parameters for statements that you create with `DBActivateSQL` or `DBNewSQLStatement`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL` or `DBNewSQLStatement`.
parameterName	char *	Name of the parameter. Use the empty string `""` to create an unnamed parameter.
parameterDirection	`tDBParameterDirection`	Direction of the parameter: input, output, input/output, or return value.
initialValue	short integer	Short integer initial value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBCreateParamShort` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Input Parameter Example

```
hstmt = DBPrepareSQL (hdbc, "SELECT UUT_NUM, LOOPNUM MEAS1,\
        MEAS2 FROM REC1000 WHERE LOOPNUM > ?");
resCode = DBCreateParamShort(hstmt, "", DB_PARAM_INPUT, 100);
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
hstmt = DBDeactivateSQL(hstmt);
```

Output Parameter Example

```
/* This example works with Microsoft SQL Server.  */
/* Create a stored procedure with input, output  */
/* and return value parameters.                  */
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest( \
        @InParam int, @OutParam int OUTPUT ) as select \
        @OutParam =  @InParam + 10 SELECT * FROM Authors \
        WHERE State <> 'CA' return @OutParam +10");

/* Set command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_STORED_PROC);
/* Prepare the SQL statement. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
/* Set the command type attribute back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                      DB_COMMAND_UNKNOWN);

/* Create the three parameters. */
resCode = DBCreateParamShort(hstmt, "",
                  DB_PARAM_RETURN_VALUE, -1);
resCode = DBCreateParamShort(hstmt, "InParam",
                  DB_PARAM_INPUT, 10);
resCode = DBCreateParamShort(hstmt, "OutParam",
                  DB_PARAM_OUTPUT, -1);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by stored procedure. */
}
/* Close the statement.  The output parameters are */
/* invalid until you close the statement.          */
resCode = DBClosePreparedSQL(hstmt);
```

```
/* Examine the parameters. */
resCode = DBGetParamShort(hstmt, 0, &retParam);
resCode = DBGetParamShort(hstmt, 1, &inParam);
resCode = DBGetParamShort(hstmt, 2, &outParam);

hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamShort, DBGetParamShort, DBClosePreparedSQL

DBCreateRecord

```
int status = DBCreateRecord (int statementHandle);
```

Purpose

Creates buffer for use with a new record. Initially sets all column values to null. You can place values into the buffer with the `DBPutCol` functions or by copying the values into bound variables. You can then insert the record in the database by calling `DBPutRecord`.

Note ***After you create a record, some database systems do not allow you to close the SQL statement until you call either*** `DBPutRecord` ***or*** `DBCancelRecordChanges`**.**

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBActivateSQL`, `DBActivateMap`, `DBNewSQLStatement`, or `DBPrepareSQL` returned.

Return Value

Name	Type	Description
status	integer	Result code that `DBCreateRecord` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
char serNum[11];
long serNumLen;
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
serialNumLen = 11;
res_code = DBBindColChar(hstmt, 1, serNum, &serNumLen, "");
...
res_code = DBCreateRecord(hstmt, 1);
strcpy(serNum,"PDX 600R");
...
resCode = DBPutRecord(hstmt);
...
resCode = DBDeactivateSQL();
```

See Also

`DBPutRecord`, `DBBindCol` functions, `DBPutCol` functions

DBCreateTableFromMap

```
int status = DBCreateTableFromMap (int mapHandle, char tableName[]);
```

Purpose

Creates a database table based on a map. This function constructs a SQL `CREATE TABLE` statement from the map and table name and executes the statement. After calling `DBCreateTableFromMap`, you can also activate the map with `DBActivateMap` and use `DBCreateRecord` and `DBPutRecord` to fill in the table values.

Note ***Some databases work with a limited set of data types. In these cases*** `DBCreateTableFromMap` ***attempts to find a compromise data type, but might not succeed.***

Parameters

Input

Name	Type	Description
mapHandle	integer	Handle to the map that `DBBeginMap` returned.
tableName	char []	Name of the database table to create.

Return Value

Name	Type	Description
status	integer	Result code that `DBCreateTableFromMap` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hmap = DBBeginMap(hdbc);
resCode = DBMapColumnToChar(hstmt, "SER_NUM", 11, serialNum,
                  &sNumStatus, "");
resCode = DBMapColumnToDouble(hmap, "MEAS1", &measurement,
                              &measStatus);
...
resCode = DBCreateTableFromMap(hmap, "TESTLOG");
```

See Also

DBBeginMap, DBActivateMap, DBDeactivateMap, DBMapColumnTo functions

DBDatabases

```
int status = 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. Each record contains two columns:

Column	Type	Description
Database	Char (128)	Database name.
Remarks	Char (256)	Remarks (can be null).

Parameter

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returned.

Return Value

Name	Type	Description
statementHandle	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***

```
DBSetBackwardCompatibility(110);
```

Example

```
hstmt = DBDatabases(hdbc);
resCode = DBBindColChar(hstmt, 1, 32, dbName,
              &dbNameStat, "");
resCode = DBBindColChar(hstmt, 2, remarks, &remarksStat,"");
...
/* Fetch records. */
...
resCode = DBDeactivateSQL(hstmt);
```

See Also

DBActivateSQL, DBActivateMap

DBDeactivateMap

```
int status = DBDeactivateMap (int mapHandle);
```

Purpose

Ends activation of a map. It is important to call `DBDeactivateMap` to free system resources.

Parameter

Input

Name	Type	Description
mapHandle	integer	Handle to the map that `DBActivateMap` returned.

Return Value

Name	Type	Description
status	integer	Result code that `DBDeactivateMap` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hmap = DBBeginMap(hdbc);
resCode = DBMapColumnToChar(hstmt, "ser_num", 11, serialNum,
                            &sNumStatus, "");
resCode = DBMapColumnToDouble(hmap, "measurement", &measurement,
                              &measStatus);
...
hstmt = DBActivateMap(map, "testlog");
while (DBFetchNext(hstmt) == 0) {
    ...
}
resCode = DBDeactivateMap(hmap);
```

See Also

`DBBeginMap`, `DBActivateMap`

DBDeactivateSQL

```
int status = DBDeactivateSQL (int statementHandle);
```

Purpose

Deactivates a SQL statement. It is important to call `DBDeactivateSQL` to free system resources. Calling `DBDeactivateSQL` is equivalent to calling `DBCloseSQLStatement` and then `DBDiscardSQLStatement`.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBActivateSQL`, `DBNewSQLStatement`, or `DBPrepareSQL` returned.

Return Value

Name	Type	Description
status	integer	Result code that `DBDeactivateSQL` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hdbc = DBConnect("DSN=CVI32_Samples");
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
...
res_code = DBDeactivateSQL(hstmt);
res_code = DBDisconnect(hdbc);
```

See Also

`DBActivateSQL`, `DBCloseSQLStatement`, `DBDiscardSQLStatement`

DBDeleteRecord

```
int status = DBDeleteRecord (int statementHandle);
```

Purpose

Deletes the current record.

Note ***After you delete the current record, your cursor remains in the position of the deleted record until you move to a different record. You must call one of the*** `DBFetch` ***functions to position the cursor on a valid record.***

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBActivateSQL`, `DBActivateMap`, `DBNewSQLStatement`, or `DBPrepareSQL` returned.

Return Value

Name	Type	Description
status	integer	Result code that `DBDeleteRecord` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
...
/* Fetch the first record and delete it. */
resCode = DBFetchNext(hstmt);
resCode = DBDeleteRecord(hstmt);
...
resCode = DBDeactivateSQL();
```

See Also

`DBActivateSQL`, `DBDeactivateSQL`, `DBFetch` functions

DBDiscardConnection

```
int status = DBDiscardConnection (int connectionHandle);
```

Purpose

Discards the specified connection. Calling `DBCloseConnection` and then `DBDiscardConnection` is equivalent to calling `DBDisconnect`.

Parameter

Input

Name	Type	Description
connectionHandle	integer	Connection handle from `DBConnect` or `DBNewConnection`.

Return Value

Name	Type	Description
status	integer	Result code that `DBDiscardConnection` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hdbc = DBNewConnection();
resCode = DBSetConnectionAttribute(hdbc,
        ATTR_DB_CONN_CONNECTION_TIMEOUT, 100);
resCode = DBSetConnectionAttribute(hdbc,
        ATTR_DB_CONN_ISOLATION_LEVEL, DB_ISOLATION_LEVEL_ISOLATED);
resCode = DBOpenConnection(hdbc);
...
resCode = DBCloseConnection(hdbc);
resCode = DBDiscardConnection(hdbc);
```

See Also

`DBDisconnect`, `DBNewConnection`, `DBOpenConnection`, `DBSetConnectionAttribute`, `DBGetConnectionAttribute`, `DBCloseConnection`

DBDiscardSQLStatement

```
int status = DBDiscardSQLStatement (int statementHandle);
```

Purpose

Discards a statement opened with DBOpenSQLStatement, DBPrepareSQL, or DBActivateSQL. Calling DBCloseSQLStatement and then DBDiscardSQLStatement is equivalent to calling DBDeactivateSQL.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that DBNewSQLStatement or DBActivateSQL returned.

Return Value

Name	Type	Description
status	integer	Result code that DBDiscardSQLStatement returns. DBDiscardSQLStatement returns the set of result codes listed in the function description for DBError.

Example

```
hstmt = DBNewSQLStatement (hdbc, "SELECT UUT_NUM, MEAS1,\
                                  MEAS2 FROM TESTRES");
resCode = DBSetStatementAttribute(hstmt,ATTR_DB_STMT_MAX_RECORDS, 1);
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_CACHE_SIZE,
                                  10);
resCode = DBOpenSQLStatement(hstmt);
...
resCode = DBGetStatementAttribute(hstmt,ATTR_DB_STMT_RECORD_COUNT,
                                  &recordCount
...
resCode = DBCloseSQLStatement(hstmt);
resCode = DBDiscardSQLStatement(hstmt);
```

See Also

DBNewSQLStatement, DBOpenSQLStatement, DBCloseSQLStatement, DBSetStatementAttribute, DBGetStatementAttribute

DBDisconnect

```
int status = DBDisconnect (int connectionHandle);
```

Purpose

Closes a connection to a database system. You should close all connections before your program terminates to free system resources. Calling `DBDisconnect` is equivalent to calling `DBCloseConnection` and then `DBDiscardConnection`. `DBDisconnect` deactivates any active maps or SQL statements on the connection.

Parameter

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection previously that `DBConnect` returned.

Return Value

Name	Type	Description
status	integer	Result code that `DBDisconnect` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hdbc = DBConnect("DSN=CVI32_Samples");
...
res_code = DBDisconnect(hdbc);
```

See Also

`DBConnect`

DBError

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

Purpose

Returns the result code of the last SQL Toolkit Library function you called. You should call `DBError` immediately after calling any other SQL Toolkit Library function that does not return a result code (for example, `DBColumnName` and `DBColumnWidth`).

Return Value

Name	Type	Description
errorCode	integer	Result code of the last SQL Toolkit Library function call.

Return Codes

Constant	Value	Description
DB_ODBC_ERROR	-12	ODBC error in DBSources.
DB_AUTOMATION_ERROR	-11	Error detected in OLE automation.
DB_DBSYS_ERROR	-10	Error detected by ADO, ODBC driver, or underlying database.
DB_EOF	-5	EOF. DBFetchNext, DBFetchPrev, or DBFetchRandom returns this value when there is no record to return.
DB_USER_CANCELED	-4	User canceled out of the logon dialog box.
DB_OUT_OF_MEMORY	-3	Windows is out of memory. This error is usually fatal.
DB_SUCCESS	0	Success.
DB_SUCCESS_WITH_INFO	1	Success with information (warning).
DB_NO_DATA_WITH_INFO	2	EOF with additional information (usually ESC during a fetch).

See Also

DBWarning, DBErrorMessage, DBNativeError

DBErrorMessage

```
char *errorString = DBErrorMessage (void);
```

Purpose

Returns the text of the error or warning generated by the last SQL Toolkit Library function you called. Any message from the underlying database system is included in the returned string.

Return Value

Name	Type	Description
errorString	integer	Pointer to a buffer containing the error or warning message text. The toolkit overwrites the buffer the next time you call a toolkit function. You must make a copy of the string if you need to call another toolkit function before you finish using the string.

See Also

DBError, DBNativeError

DBExecutePreparedSQL

```
int status = DBExecutePreparedSQL (int statementHandle);
```

Purpose

Executes a stored procedure or SQL statement that you have prepared with `DBPrepareSQL`.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned.

Return Value

Name	Type	Description
status	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam int, @OutParam int OUTPUT ) as  \
        select @OutParam = @InParam + 10 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam +10");

/* Prepare a statement that calls the stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamShort(hstmt, 1, 10);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
```

```
    /* Process records returned by the stored procedure. */
}
/* Close the statement.  Output parameters are invalid. */
/* until you close the statement.                       */
resCode = DBClosePreparedSQL(hstmt);
/* Examine the parameter values. */
resCode = DBGetParamShort(hstmt, 0, &retParam);
resCode = DBGetParamShort(hstmt, 1, &inParam);
resCode = DBGetParamShort(hstmt, 2, &outParam);
/* Discard the statement. */
hstmt = DBDiscardSQLStatement(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBClosePreparedSQL

DBFetchNext

```
int status = DBFetchNext (int statementHandle);
```

Purpose

Retrieves the next record from the database. This function places the column values in any variables that you previously specified using the functions for binding or mapping variables. When DBFetchNext reaches the last record that the SELECT statement returns, it returns an end-of-file result, DB_EOF.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from DBActivateSQL, DBActivateMap, or any of the functions that return a statement handle.

Return Value

Name	Type	Description
status	integer	Result code that DBFetchNext returns. DBFetchNext returns the set of result codes listed in the function description for DBError. Returns a value of DB_EOF when DBFetchNext reaches the last record returned by the SELECT statement.

Example

```
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
...
while (DBFetchNext(hstmt) == 0) {
    ...
}
resCode = DBDeactivateSQL();
```

See Also

DBFetchPrev, DBFetchRandom, DBActivateMap, DBActivateSQL, DBBindCol functions, DBAllowFetchAnyDirection

DBFetchPrev

```
int status = DBFetchPrev (int statementHandle);
```

Purpose

Retrieves the previous record from the database. This function places the column values in any variables that you previously specified using the functions for binding or mapping variables.

You cannot use this function if you are using a forward-only cursor. For more information on cursors, refer to `ATTR_DB_STMT_CURSOR_TYPE` in the function description for `DBSetAttributeDefault` or `DBSetStatementAttribute`. When `DBFetchPrev` attempts to fetch a record before the first record returned by the `SELECT` statement, it returns a result of `DB_EOF`.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL`, `DBActivateMap`, or any of the functions that return a statement handle.

Return Value

Name	Type	Description
status	integer	Result code that `DBFetchPrev` returns. `DBFetchPrev` returns the set of result codes listed in the function description for `DBError`. Returns a value of `DB_EOF` when attempting to fetch a record before the first record returned by the `SELECT` statement.

Example

```
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_CURSOR_TYPE,
                                DB_CURSOR_TYPE_KEYSET);
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
...
resCode = DBFetchNext(hstmt)
/* This step is repeated to read other records. */
...
resCode = DBFetchPrev(hstmt);
...
resCode = DBDeactivateSQL();
```

See Also

DBActivateSQL, DBActivateMap, DBFetchNext, DBFetchRandom, DBBindCol functions, DBMapColumnTo functions, DBPutRecord, DBSetAttributeDefault

DBFetchRandom

```
int status = DBFetchRandom (int statementHandle, long recordNumber);
```

Purpose

Retrieves the designated record from the database. This function places the column values in any variables that you previously specified using the functions for binding or mapping variables. You cannot use this function if you are using a forward-only cursor. For more information on cursors, refer to `ATTR_DB_STMT_CURSOR_TYPE` in the function description for `DBSetAttributeDefault` or `DBSetStatementAttribute`. 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`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL`, `DBNewSQLStatement`, or any of the functions that return a statement handle.
recordNumber	long integer	Record number to fetch. The first record is `1`.

Return Value

Name	Type	Description
status	integer	Result code that `DBFetchRandom` returns. `DBFetchRandom` returns the set of result codes listed in the function description for `DBError`. Returns `DB_EOF` if the designated record is not in the result set.

Example

```
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_CURSOR_TYPE,
                                DB_CURSOR_TYPE_KEYSET);
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
...
numRecs = DBNumberOfRecords(hstmt)
/* Fetch the last record. */
resCode = DBFetchRandom(hstmt,numRecs);
...
resCode = DBDeactivateSQL();
```

See Also

DBActivateSQL, DBActivateMap, DBFetchNext, DBFetchPrev, DBBindCol functions, DBMapColumn functions, DBPutRecord, DBSetAttributeDefault

DBForeignKeys

```
int status = DBForeignKeys (int connectionHandle,
                            char primaryKeyTableName[],
                            char foreignKeyTableName[]);
```

Purpose

Creates and activates a `SELECT` statement that returns information about the set of columns that make up the foreign keys of a table. You can then use the `DBFetch` and `DBBindCol` or `DBGetCol` functions to retrieve the information. Each record contains the columns shown in the following table.

☞ **Note** ***Not all database systems support foreign keys.***

Column	Type	Description
PK Table Catalog	string	Primary key table catalog.
PK Table Schema	string	Primary key table schema.
PK Table Name	string	Primary key table name.
PK Column Name	string	Primary key column name.
PK Column GUID	integer	Primary key column GUID.
PK Column Property ID	integer	Primary key column property ID.
FK Table Catalog	string	Foreign key table catalog.
FK Table Schema	string	Foreign key table schema.
FK Table Name	string	Foreign key table name.
FK Column Name	string	Foreign key column name.
FD Column GUID	integer	Foreign key GUID.
FK Column Property ID	integer	Foreign key column property ID.
Sequence Number	short integer	Number of this column within the foreign key.

Column	Type	Description

Update Action	short integer	Action applied to the foreign key when an `UPDATE` is performed.
		`DB_CASCADE` = `0`
		`DB_RESTRICT` = `1`
		`DB_SET_NULL` = `2`

Delete Action	short integer	Action applied to the foreign key when you perform a `DELETE`.
		`DB_CASCADE` = `0`
		`DB_RESTRICT` = `1`
		`DB_SET_NULL` = `2`

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returned.
primaryKeyTableName	char []	Pointer to a string containing the name of the table about which you want to select primary key information.
foreignKeyTableName	char []	Pointer to a string containing the name of the table about which you want to select foreign key information.

Return Value

Name	Type	Description
statementHandle	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***

```
DBSetBackwardCompatibility(110);
```

Example

```
hstmt = DBForeignKeys(hdbc, "testpk", "testfk);

resCode = DBBindColChar (hstmt, 1, 128, pkTableCatalog, &stat1, "");
resCode = DBBindColChar (hstmt, 2, 128, pkTableSchema, &stat2, "");
...
resCode = DBBindColShort (hstmt, 15, 128, deleteAction, &stat15);

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    printf("table Catalog %s Schema %s name %s \n", pkTableCatalog,
    pkTableSchema, pkTableName);
}
resCode = DBDeactivateSQL(hstmt);
```

DBFree

```
void DBFree (void *memBlockPointer);
```

Purpose

Causes deallocation of the memory that Mem Block Pointer points to, that is, it makes the memory unavailable for further use. In an external compiler environment, there can be one set of `malloc()` and `free()` functions used by the LabWindows/CVI SQL toolkit and an entirely different set used by the external compiler. Use `DBFree` to ensure that the proper function frees the memory that the LabWindows/CVI SQL Toolkit allocates and returns.

Note ***Do not use*** `DBFree` ***to free items that you extract from variants. For information on functions you can use to free resources, refer to Chapter 11,*** *ActiveX Automation Library****, of the*** *LabWindows/CVI Standard Libraries Reference Manual****.***

Parameter

Input

Name	Type	Description
memBlockPointer	void pointer	Pointer to the memory block to deallocate. If you enter a null pointer into this control, no action occurs.

DBFreeVariantArray

```
int DBFreeVariantArray (VARIANT *recordsArray, int clearMemberVariants,
                        unsigned int recordsinArray, unsigned int
                        fieldsinArray);
```

Purpose

Frees the variant array that `DBGetVariantArray` returned.

Parameters

Input

Name	Type	Description
recordsArray	VARIANT *	The array of variants from `DBGetVariantArray`.
clearMemberVariants	integer	Specifies whether to free any memory allocated within the variants before freeing the array. You must free the allocated memory in the variants to prevent memory leaks.
recordsinArray	unsigned integer	The number of records/rows in the array. Use the Records Returned value from `DBGetVariantArray` for this value.
fieldsinArray	unsigned integer	The number of fields/columns in the array. Use the Fields Returned value from `DBGetVariantArray` for this value.

Return Value

Name	Type	Description
status	integer	Result code that `DBFreeVariantArray` returns. This function returns the set of result codes listed in the function description for `DBError`.

Example

```
/* Execute a select statement */
hstmt = DBActivateSQL (hdbc, "SELECT * FROM TESTRES");

resCode = DBGetVariantArray(hstmt, &cArray, &numRecs, &numFields);
for (i = 0; i < numRecs; i++) {
```

```
        for (j = 0; j < numFields; j++) {
            resCode = DBGetVariantArrayValue(cArray, numRecs,
                                             numFields, CAVT_CSTRING,
                                             i, j, &tempStr);
            if (resCode == DB_NULL_DATA) {
                /* Handle null data. */
            } else {
                /* Handle other data. */
                DBFree(tempStr);
            }
        }
    }
    resCode = DBFreeVariantArray(cArray, 1, numRecs, numFields);
```

See Also

DBGetVariantArray, DBGetVariantArrayColumn, DBGetVariantArrayValue

DBGetColBinary

```
int status = DBGetColBinary (int statementHandle, int columnNumber, void
                             **value);
```

Purpose

Gets a binary value from the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBGetColBinary`. You must use `DBFree` to free the value.

☞ **Note** ***You can use*** `DBGetColBinary` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record that is the source of the value. The first column number is `1`.

Output

Name	Type	Description
value	void pointer (passed by reference)	Variable that receives the binary value from the specified field/column in the record. You must use `DBFree` to free the value.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColBinary` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
unsigned char* fromDBBits = NULL;

hstmt = DBActivateSQL(hdbc, "SELECT THE BITS FROM BINTEST");

while ((dbStatus = DBFetchNext(hstmt) == DB_SUCCESS) {
    dbStatus = DBGetColBinary(hstmt, 1, (void **)(&fromDBBits));
    if (dbStatus == DB_NULL_DATA) {
       /* Processing for NULL data. */
    }
    ...
DBFree(fromDBBits);
}
dbStatus = DBDeactivateSQL(hstmt);
hstmt = 0;
```

See Also

DBGetColBinaryBuffer, DBBindColBinary, DBMapColumnToBinary

DBGetColBinaryBuffer

```
int status = DBGetColBinaryBuffer (int statementHandle, int columnNumber,
                                   void *value, int bufferLength);
```

Purpose

Gets a binary value from the current record into a buffer. Instead of binding values and then fetching a record, you can fetch a record and then use `DBGetColBinaryBuffer`.

Note ***You can use*** `DBGetColBinaryBuffer` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record that is the source of the value. The first column number is `1`.
bufferLength	integer	Length of the buffer.

Output

Name	Type	Description
value	void pointer	Buffer that receives the value from the specified field/column in the record.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColBinaryBuffer` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
unsigned char fromDBBits[6];

hstmt = DBActivateSQL(hdbc, "SELECT THE BITS FROM BINTEST");

while ((dbStatus = DBFetchNext(hstmt)) == DB_SUCCESS) {
    dbStatus = DBGetColBinaryBuffer(hstmt, 1, fromDBBits, bitsSize);
if (dbStatus == DB_NULL_DATA) {
        /* Handle NULLs. */
    }
    /* Use the value. */
}
dbStatus = DBDeactivateSQL(hstmt);
hstmt = 0;
```

See Also

DBBindColChar, DBMapColumnToChar

DBGetColChar

```
int status = DBGetColChar (int statementHandle, int columnNumber, char
                           **value, char *formatString);
```

Purpose

Gets a string value from the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBGetColChar`. You must use `DBFree` to free the string.

Note ***You can use*** `DBGetColChar` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record that is the source of the value. The first column number is `1`.
formatString	char []	Format string. Use the empty string, `""`, if you want the default format. See Appendix C, *Format Strings*, for details about formatting.

Output

Name	Type	Description
value	char * (passed by reference)	Variable that receives the value from the specified field/column in the record. Use `DBFree` to free the string.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColChar` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, CHANGER, \
    LOOPNUM, MEAS1, MEAS2 FROM REC1000");

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
/* Get values into the record. */
    resCode = DBGetColChar(hstmt, 1, &uutNum, "");
    resCode = DBGetColInt(hstmt, 2, &changer);
    resCode = DBGetColFloat(hstmt, 3, &meas1);
    resCode = DBGetColDouble(hstmt, 4, &meas2);
    resCode = DBGetColShort(hstmt, 5, &loopNum);
    /* Process values. */
    ...
    DBFree(uutNum);
         }

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

`DBBindColChar`, `DBMapColumnToChar`

DBGetColCharBuffer

```
int status = DBGetColCharBuffer (int statementHandle, int columnNumber, char
                                 value[], int bufferLength, char *formatString);
```

Purpose

Gets a string value from the current record into a buffer. Instead of binding values and then fetching a record, you can fetch a record and then use `DBGetColCharBuffer`.

Note ***You can use*** `DBGetColCharBuffer` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record that is the source of the value. The first column number is `1`.
bufferLength	integer	Length of the buffer.
formatString	char []	Format string. Use the empty string, `" "`, if you want the default format. See Appendix C, *Format Strings*, for details about formatting.

Output

Name	Type	Description
value	char []	Variable that receives the value from the specified field/column in the record.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColCharBuffer` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, CHANGER, \
    LOOPNUM, MEAS1, MEAS2 FROM REC1000");

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
/* Get values into the record. */
    resCode = DBGetColCharBuffer(hstmt, 1, uutNum, 10, "");
    ...
    /* Process values. */
    ...
         }

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColChar, DBMapColumnToChar

DBGetColDouble

```
int status = DBGetColDouble (int statementHandle, int columnNumber,
                             double *value);
```

Purpose

Gets a double-precision value from the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBGetColDouble`.

Note ***You can use*** `DBGetColDouble` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record that is the source of the value. The first column number is `1`.

Output

Name	Type	Description
value	double-precision (passed by reference)	Short integer variable that receives the value from the specified field/column in the record.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColDouble` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, CHANGER, \
    LOOPNUM, MEAS1, MEAS2 FROM REC1000");

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
/* Get values into the record. */
    resCode = DBGetColChar(hstmt, 1, &uutNum, "");
    resCode = DBGetColInt(hstmt, 2, &loopNum);
    resCode = DBGetColFloat(hstmt, 3, &meas1);
    resCode = DBGetColDouble(hstmt, 4, &meas2);
    resCode = DBGetColShort(hstmt, 5, &changer);
    /* Process values. */
    ...
    DBFree(uutNum);
         }

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColDouble, DBMapColumnToDouble

DBGetColFloat

```
int status = DBGetColFloat (int statementHandle, int columnNumber,
                            float *value);
```

Purpose

Gets a floating-point value from the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBGetColFloat`.

Note ***You can use*** `DBGetColFloat` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record that is the source of the value. The first column number is `1`.

Output

Name	Type	Description
value	float (passed by reference)	Short integer variable that receives the value from the specified field/column in the record.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColFloat` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, CHANGER, \
    LOOPNUM, MEAS1, MEAS2 FROM REC1000");

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
/* Get values into the record. */
    resCode = DBGetColChar(hstmt, 1, &uutNum, "");
    resCode = DBGetColInt(hstmt, 2, &loopNum);
    resCode = DBGetColFloat(hstmt, 3, &meas1);
    resCode = DBGetColDouble(hstmt, 4, &meas2);
    resCode = DBGetColShort(hstmt, 5, &changer);
    /* Process values. */
    ...
    DBFree(uutNum);
         }

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColFloat, DBMapColumnToFloat

DBGetColInt

```
int status = DBGetColInt (int statementHandle, int columnNumber, int *value);
```

Purpose

Gets an integer value from the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBGetColInt`.

Note ***You can use*** `DBGetColInt` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record that is the source of the value. The first column number is `1`.

Output

Name	Type	Description
value	integer (passed by reference)	Short integer variable that receives the value from the specified field/column in the record.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColInt` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, CHANGER, \
    LOOPNUM, MEAS1, MEAS2 FROM REC1000");

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
/* Get values from the current record. */
    resCode = DBGetColChar(hstmt, 1, &uutNum, "");
    resCode = DBGetColInt(hstmt, 2, &changer);
    resCode = DBGetColFloat(hstmt, 3, &meas1);
    resCode = DBGetColDouble(hstmt, 4, &meas2);
    resCode = DBGetColShort(hstmt, 5, &loopNum);
    /* Process values. */
    ...
    DBFree(uutNum);
         }

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColInt, DBMapColumnToInt

DBGetColShort

```
int status = DBGetColShort (int statementHandle, int columnNumber, short
                            *value);
```

Purpose

Gets a short integer value from the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBGetColShort`.

Note ***You can use*** `DBGetColShort` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record that is the source of the value. The first column number is `1`.

Output

Name	Type	Description
value	short integer (passed by reference)	Short integer variable that receives the value from the specified field/column in the record.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColShort` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, CHANGER, \
    LOOPNUM, MEAS1, MEAS2 FROM REC1000");

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
/* Get values into the record. */
    resCode = DBGetColChar(hstmt, 1, &uutNum, "");
    resCode = DBGetColInt(hstmt, 2, &changer);
    resCode = DBGetColFloat(hstmt, 3, &meas1);
    resCode = DBGetColDouble(hstmt, 4, &meas2);
    resCode = DBGetColShort(hstmt, 5, &loopNum);
    /* Process values. */
    ...
    DBFree(uutNum);
         }

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColShort, DBMapColumnToShort

DBGetColumnAttribute

```
int status = DBGetColumnAttribute (int statementHandle, int index,
                                   tDBColumnAttr attribute, void *value);
```

Purpose

Obtains a field/column attribute.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
index	integer	Index of the column. The index of the first column is `1`.
attribute	`tDBColumnAttr`	Attribute to get.

Output

Name	Type	Description
value	any type (passed by reference)	Value for the parameter attribute. The type of the value varies depending on the attribute. To free the strings that this function returns, use `DBFree`. To free allocated memory within the variants this function returns, use `CA_VariantClear`. For information on functions you can use to free resources, refer to Chapter 11, *ActiveX Automation Library*, of the *LabWindows/CVI Standard Libraries Reference Manual*.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColumnAttribute` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion for attribute and value Parameters

Attribute	Type	Description
`ATTR_DB_COLUMN_VALUE`	`VARIANT`	Value of field/column.
`ATTR_DB_COLUMN_ORIGINAL_VALUE`	`VARIANT`	Original value of field/column.
`ATTR_DB_COLUMN_UNDERLYING_VALUE`	`VARIANT`	Value of field/column currently in the database, which might have changed since you read the record.
`ATTR_DB_COLUMN_NAME`	char*	Name of parameter
`ATTR_DB_COLUMN_PRECISION`	byte	Total number of digits.
`ATTR_DB_COLUMN_NUMERIC_SCALE`	byte	Number of digits to the right of decimal.
`ATTR_DB_COLUMN_DEFINED_SIZE`	long integer	Defined size in bytes.
`ATTR_DB_COLUMN_ACTUAL_SIZE`	long integer	Actual size reserved in bytes.

Attribute	Type	Description
ATTR_DB_COLUMN_ATTRIBUTES	long integer	Sum of zero or more of the following values: DB_COLUMN_MAY_DEFER DB_COLUMN_UPDATABLE DB_COLUMN_UNKNOWN_UPDATEABLE DB_COLUMN_FIXED DB_COLUMN_IS_NULLABLE DB_COLUMN_MAY_BE_NULL DB_COLUMN_LONG DB_COLUMN_ROW_ID DB_COLUMN_ROW_VERSION DB_COLUMN_CACHE_DEFFERED
ATTR_DB_COLUMN_TYPE	long integer	Type of parameter: DB_EMPTY DB_TINYINT DB_SMALLINT DB_INTEGER DB_BIGINT DB_FLOAT DB_UNSIGNEDTINYINT DB_UNSIGNEDSMALLINT DB_UNSIGNEDINT DB_UNSIGNEDBIGINT DB_DOUBLEPRECISION DB_CURRENCY DB_DECIMAL DB_NUMERIC DB_BOOLEAN DB_ERROR DB_USERDEFINED DB_VARIANT DB_IDDISPATCH DB_IUNKNOWN DB_GUID DB_DBDATE DB_DBTIME DB_DATETIME DB_DATE DB_BSTR DB_CHAR DB_VARCHAR DB_LONGVARCHAR DB_WCHAR DB_VARWCHAR DB_LONGVARWCHAR DB_BINARY DB_VARBINARY DB_LONGVARBINARY

Example

```
resCode = DBGetColumnAttribute (hstmt, i, ATTR_DB_COLUMN_NAME,
                                &tempStr);
DBFree(tempStr);
resCode = DBGetColumnAttribute (hstmt, i, ATTR_DB_COLUMN_VALUE,
                                &valueVariant);
resCode = CA_VariantConvertToType (&valueVariant,
                                   CAVT_CSTRING, &tempStr);
CA_FreeMemory(tempStr);
```

DBGetColVariant

```
int status = DBGetColVariant (int statementHandle, int columnNumber,
                              VARIANT *value);
```

Purpose

Gets a value from the current record as a Variant. Variants are useful for data types beyond the traditional integer, short, floating-point, double-precision, and string data types.

Note ***You can use*** `DBGetColVariant` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record that is the source of the value. The first column number is `1`.

Output

Name	Type	Description
value	`VARIANT` (passed by reference)	Variant variable that receives the value from the specified field/column in the record. Use `CA_VariantClear` to free memory within the variant when you no longer need it.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetColVariant` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
VARIANT loopNumV
...
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, CHANGER, \
    LOOPNUM, MEAS1, MEAS2 FROM REC1000");
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
    resCode = DBGetColVariant(hstmt, 2, &loopNumV);
    vStatus = CA_VariantConvertToType (&loopNumV, CAVT_INT,
                                       &loopNum);
    ...
         }
/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColFloat, DBMapColumnToFloat

DBGetConnectionAttribute

```
int status = DBGetConnectionAttribute (int connectionHandle,
                          tDBConnectionAttr attribute, void *value);
```

Purpose

Obtains the value of a connection attribute.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the connection that `DBNewConnection` or `DBConnect` returned.
attribute	`tDBConnectionAttr`	Attribute to get. Some providers do not support all attributes.

Output

Name	Type	Description
value	any type (passed by reference)	Value for the attribute. The type of the value varies depending on the attribute. Some providers do not support all attributes. To free the strings that this function returns, use `DBFree`. To free allocated memory within the variants this function returns, use `CA_VariantClear`. For information on functions you can use to free resources, refer to Chapter 11, *ActiveX Automation Library*, of the *LabWindows/CVI Standard Libraries Reference Manual*.

Parameter Discussion for attribute and value Parameters

Attribute	Type	Description
ATTR_DB_CONN_CONNECTION_STRING	string	A series of argument = value clauses separated by semicolons that describe the connection. The Active Data Object (ADO) standard recognizes the following arguments. All other arguments are defined by the provider. Provider—Name of the provider to use for the connection. Data Source—Name of the data source for the connection. User ID—User name to use when opening the connection. Password—Password to use when opening the connection. File Name—Name of a provider-specific file (for example, a persisted data source object) containing preset connection information. Remote Provider—Name of a provider to use when opening a client-side connection. Remote Server—Path name of the server to use when opening a client-side connection.
ATTR_DB_CONN_COMMAND_TIMEOUT	long integer	Number of seconds to wait for a command to execute
ATTR_DB_CONN_CONNECTION_TIMEOUT	long integer	Number of seconds to wait for a connection to be established.

Attribute	Type	Description
ATTR_DB_CONN_DEFAULT_DATABASE	string	Name of the default database for database systems that support storing tables in multiple databases.
ATTR_DB_CONN_ISOLATION_LEVEL	long integer	Isolation level of the connection: DB_ISOLATION_LEVEL_UNSPECIFIED—Provider is using a different isolation level than the level specified by the user and the toolkit cannot determine that level. DB_ISOLATION_LEVEL_CHAOS—You cannot overwrite pending changes from more highly isolated transactions. DB_ISOLATION_LEVEL_READ_UNCOMMITTED—From one transaction you can view uncommitted changes in other transactions. DB_ISOLATION_LEVEL_READ_COMMITTED—Default. From one transaction you can view changes in other transactions only after they have been committed. DB_ISOLATION_LEVEL_REPEATABLE_READ—From one transaction you cannot see changes made in other transactions, but a new query can bring new recordsets. DB_ISOLATION_LEVEL_SERIALIZABLE—Transactions take place in isolation of other transactions.

Attribute	Type	Description
ATTR_DB_CONN_ATTRIBUTES	long integer	Attributes of the connection, the sum of one or more of the following values: DB_XACT_COMMIT_RETAINING—Performs retaining commits. In other words, calling DBCommit automatically starts a new transaction. Not all providers support this value. DB_XACT_ABORT_RETAINING—Performs retaining aborts. In other words, calling DBRollback automatically starts a new transaction. Not all providers support this value.
ATTR_DB_CONN_CURSOR_LOCATION	long integer	Location of the cursor: DB_CURSOR_LOC_NONE—Cursor location has not been set or cannot be determined. DB_CURSOR_LOC_SERVER—Default. Uses cursors that the data provider or driver supply. DB_CURSOR_LOC_CLIENT—Uses client-side cursors supplied by a local cursor library.

Attribute	Type	Description
ATTR_DB_CONN_MODE	long integer	Connection mode: DB_CONN_MODE_UNKNOWN—Permissions are not set or cannot be determined. DB_CONN_MODE_READ—Read-only permissions. DB_CONN_MODE_WRITE—Write-only permissions. DB_CONN_MODE_READ_WRITE—Read/write permissions. DB_CONN_MODE_SHARE_DENY_READ—Prevents others from opening connection with read permissions. DB_CONN_MODE_SHARE_DENY_WRITE—Prevents others from opening connection with write permissions. DB_CONN_MODE_SHARE_EXCLUSIVE—Prevents others from opening connection. DB_CONN_MODE_SHARE_DENY_NONE—Prevents others from opening connection with any permissions.
TTR_DB_CONN_PROVIDER	string	Name of the provider of the connection. The default is MSDASQL, the ODBC provider.
ATTR_DB_CONN_STATE	long integer	Open/closed state of the connection: DB_OBJECT_STATE_CLOSED = 0 DB_OBJECT_STATE_OPEN = 1
ATTR_DB_CONN_CONNECTION_OBJECT	CAObjHandle	The ActiveX object handle of the connection.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetConnectionAttribute` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
int timeout, isoLevel;
hdbc = DBConnect("DSN=cvi ss;User ID=sa;Password=");
resCode = DBGetConnectionAttribute(hdbc,
        ATTR_DB_CONN_CONNECTION_TIMEOUT, &timeout);
resCode = DBGetConnectionAttribute(hdbc,
        ATTR_DB_CONN_ISOLATION_LEVEL, &isoLevel
```

See Also

`DBConnect`, `DBNewConnection`, `DBOpenConnection`, `DBSetConnectionAttribute`

DBGetParamAttribute

```
int status = DBGetParamAttribute (int statementHandle, int index,
                                  tDBParamAttr attribute, void *value);
```

Purpose

Obtains the attribute of a parameter.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBNewSQLStatement`, or `DBActivateMap` returned.
index	integer	Index of the parameter. The index of the first parameter is `1`.
attribute	`tDBParamAttr`	Attribute to get.

Output

Name	Type	Description
value	any type (passed by reference)	Value for the parameter attribute. The type of the value varies depending on the attribute. To free the strings that this function returns, use `DBFree`. To free allocated memory within the variants this function returns, use `CA_VariantClear`. For information on functions you can use to free resources, refer to Chapter 11, *ActiveX Automation Library*, of the *LabWindows/CVI Standard Libraries Reference Manual*.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamAttribute` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion for attribute and value Parameters

Attribute	Type	Description
`ATTR_DB_PARAM_VALUE`	`VARIANT`	Value of parameter.
`ATTR_DB_PARAM_NAME`	char*	Name of parameter.

`ATTR_DB_PARAM_DIRECTION`	long integer	Direction of parameter
		`DB_PARAM_INPUT`
		`DB_PARAM_OUTPUT`
		`DB_PARAM_INPUT_OUTPUT`
		`DB_PARAM_RETURN_VALUE`
		`DB_PARAM_UNKNOWN`
		Note: ***Some providers cannot determine the direction of parameters to stored procedures. You cannot rely on*** `DBRefreshParams` ***in such cases.***

`ATTR_DB_PARAM_PRECISION`	byte	Total number of digits.
`ATTR_DB_PARAM_NUMERIC_SCALE`	byte	Number of digits to the right of decimal.
`ATTR_DB_PARAM_SIZE`	long	Maximum size in bytes.

Attribute	Type	Description
ATTR_DB_PARAM_ATTRIBUTES	long integer	Sum of zero or more of the following values: DB_PARAM_SIGNED DB_PARAM_NULLABLE DB_PARAM_LONG
ATTR_DB_PARAM_TYPE	long integer	Type of parameter: DB_EMPTY DB_TINYINT DB_SMALLINT DB_INTEGER DB_BIGINT DB_UNSIGNEDTINYINT DB_UNSIGNEDSMALLINT DB_UNSIGNEDINT DB_UNSIGNEDBIGINT DB_FLOAT DB_DOUBLEPRECISION DB_CURRENCY DB_DECIMAL DB_NUMERIC DB_BOOLEAN DB_ERROR DB_BINARY DB_VARIANT DB_IDDISPATCH DB_IUNKNOWN DB_GUID DB_DATE DB_DBDATE DB_DBTIME DB_DATETIME DB_BSTR DB_CHAR DB_VARCHAR DB_WCHAR DB_VARWCHAR DB_LONGVARCHAR DB_LONGVARWCHAR DB_VARBINARY DB_LONGVARBINARY DB_USERDEFINED

Example

```
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest( \
                                @InParam float, @OutParam \
                                float OUTPUT ) as select \
                                @OutParam = @InParam + 1.5 \
                                SELECT * FROM Authors WHERE\
                                State <> 'CA' return \
                                @OutParam +1.7");

/* Prepare a statement which calls the stored procedure */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);
resCode = DBRefreshParams(hstmt);

/* Execute the statement */
resCode = DBExecutePreparedSQL(hstmt);
resCode = DBClosePreparedSQL(hstmt);

resCode = DBGetParamAttribute (hstmt, 2, ATTR_DB_PARAM_NAME,
                               &tempStr);
printf(" name %s", tempStr);
DBFree(tempStr);

resCode = DBGetParamAttribute (hstmt, 2, ATTR_DB_PARAM_DIRECTION,
                               &tempLong);
```

DBGetParamBinary

```
int status = DBGetParamBinary (int statementHandle, int index, void **value);
```

Purpose

Obtains the value of a parameter for a stored procedure or SQL statement that you have prepared with `DBPrepareSQL`. Output parameters are invalid until you close the statement with `DBClosePreparedSQL`. You must use `DBFree` to free the returned buffer.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter. The first parameter index is `1`.

Output

Name	Type	Description
value	void pointer (passed by reference)	Binary value for the parameter. You must use `DBFree` to free the binary data when you no longer need it.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamBinary` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Output Parameter Example

```
/* This example works with Microsoft SQL Server. */
unsigned char inParam[6] = "in";
unsigned char readInParam[6];
unsigned char *outParam;
char* retParam;
...
/* Create the stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest( \
       @InParam binary(6), @OutParam binary(6) OUTPUT ) as \
       select @OutParam = @InParam SELECT * FROM Authors \
       WHERE State <> 'CA' return 10.1");

/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_CVItest");

/* Set the command type attribute back to the default  */
/* for future commands.                                */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Put some data in the input variable.             */
/* Notice that using binary allows embedded nulls.  */
inParam[0]='N';inParam[1]='C';inParam[2]='B';
inParam[3]='\0'; inParam[4]='O';inParam[5]='K';
outParam = malloc(6);
/* Create the parameters. */
resCode = DBCreateParamChar(hstmt, "", DB_PARAM_RETURN_VALUE,
                            retParam, 6);
resCode = DBCreateParamBinary(hstmt, "InParam", DB_PARAM_INPUT, "",
                              6);
resCode = DBCreateParamBinary(hstmt, "OutParam", DB_PARAM_OUTPUT, "",
                              6);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by the stored procedure. */
}
```

```
/* Close the statement.  Output values are invalid  */
/* until you close the statement.                   */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamChar(hstmt, 0, &retParam, "");
resCode = DBGetParamBinaryBuffer(hstmt, 1, readInParam, 6);
resCode = DBGetParamBinary(hstmt, 2, (void **)&(outParam));
DBFree(retParam);
DBFree(outParam);

/* Deactivate the statement. */
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamBinary, DBGetParamBinaryBuffer, DBClosePreparedSQL

DBGetParamBinaryBuffer

```
int status = DBGetParamBinaryBuffer (int statementHandle, int index, void
                          *value, unsigned int sizeinBytes);
```

Purpose

Obtains the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Output parameters are invalid until you close the statement with `DBClosePreparedSQL`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter. The first parameter index is `1`.
sizeinBytes	integer	Size of the value buffer in bytes.

Output

Name	Type	Description
value	void pointer	Buffer to hold returned binary value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamBinaryBuffer` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* This example works with Microsoft SQL Server. */
unsigned char inParam[6] = "in";
unsigned char readInParam[6];
unsigned char *outParam;
char* retParam;
...
/* Create the stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest( \
       @InParam binary(6), @OutParam binary(6) OUTPUT ) as \
       select @OutParam = @InParam SELECT * FROM Authors \
       WHERE State <> 'CA' return 10.1");

/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_CVItest");

/* Set the command type attribute back to the default */
/* for future commands.                               */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Put some data in the input variable.            */
/* notice that using binary allows embedded nulls. */
inParam[0]='N';inParam[1]='C';inParam[2]='B';
inParam[3]='\0'; inParam[4]='O';inParam[5]='K';
outParam = malloc(6);
/* Create the parameters. */
resCode = DBCreateParamChar(hstmt, "", DB_PARAM_RETURN_VALUE,
                            retParam, 6);
resCode = DBCreateParamBinary(hstmt, "InParam", DB_PARAM_INPUT, "",
                              6);
resCode = DBCreateParamBinary(hstmt, "OutParam", DB_PARAM_OUTPUT, "",
                              6);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by the stored procedure. */
}
```

```
/* Close the statement.  Output values are invalid  */
/* until you close the statement.                   */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamChar(hstmt, 0, &retParam, "");
resCode = DBGetParamBinaryBuffer(hstmt, 1, readInParam, 6);
resCode = DBGetParamBinary(hstmt, 2, (void **)&(outParam));
DBFree(retParam);
DBFree(outParam);

/* Deactivate the statement. */
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBCreateParamBinary, DBSetParamBinary, DBGetParamBinary, DBClosePreparedSQL

DBGetParamChar

```
int status = DBGetParamChar (int statementHandle, int index, char **value,
                             char *formatString);
```

Purpose

Obtains the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Output parameters are invalid until you close the statement with `DBClosePreparedSQL`. You must use `DBFree` to free the returned string.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter.The index of the first parameter is `1`.
formatString	char []	Format string. Use the empty string, `""`, if you want the default format. See Appendix C, *Format Strings*, for details about formatting.

Output

Name	Type	Description
value	char * (passed by reference)	String value for the parameter. You must use `DBFree` to free the string when you no longer need it.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamChar` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
char inParam[11] = "in";
char readInParam[11];
char outParam[11];
char* retParam;
...
/* Create the stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest( \
    @InParam char(10), @OutParam char(10) OUTPUT ) as \
    select @OutParam = 'out' SELECT * FROM Authors WHERE \
    State <> 'CA' return 10.1");
/* Set the command type attribute to store procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
/* Set command type attribute back to default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);
/* Create the parameters. */
resCode = DBCreateParamChar(hstmt, "", DB_PARAM_RETURN_VALUE, inParam,
                            10);
resCode = DBCreateParamChar(hstmt, "InParam", DB_PARAM_INPUT, "", 10);
resCode = DBCreateParamChar(hstmt, "OutParam", DB_PARAM_OUTPUT, "", 10);
/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by the stored procedure. */
}
/* To make output parameters valid you must close the statement. */
/* until you close the statement.                                */
resCode = DBClosePreparedSQL(hstmt);
/* Examine the parameter values. */
resCode = DBGetParamChar(hstmt, 0, &retParam, "");
resCode = DBGetParamCharBuffer(hstmt, 1, readInParam, 10, "");
resCode = DBGetParamCharBuffer(hstmt, 2, outParam, 10, "");
...
DBFree(retParam);
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamChar, DBClosePreparedSQL

DBGetParamCharBuffer

```
int status = DBGetParamCharBuffer (int statementHandle, int index, char
                                   value[], int bufferLength, char *formatString);
```

Purpose

Obtains the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Output parameters are invalid until you close the statement with `DBClosePreparedSQL`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter. The index of the first parameter is `1`.
bufferLength	integer	Size of the value buffer.
formatString	char []	Format string. Use the empty string, `""`, if you want the default format. See Appendix C, *Format Strings*, for details about formatting.

Output

Name	Type	Description
value	char []	Buffer to hold returned string value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamCharBuffer` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
char inParam[11] = "in";
char readInParam[11];
char outParam[11];
char* retParam;
...
/* Create the stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest( \
    @InParam char(10), @OutParam char(10) OUTPUT ) as \
    select @OutParam = 'out' SELECT * FROM Authors WHERE \
    State <> 'CA' return 10.1");

/* Set the command type attribute to store procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
/* Set command type attribute back to default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Create the parameters. */
resCode = DBCreateParamChar(hstmt, "", DB_PARAM_RETURN_VALUE,
                            inParam, 10);
resCode = DBCreateParamChar(hstmt, "InParam", DB_PARAM_INPUT, "",
                            10);
resCode = DBCreateParamChar(hstmt, "OutParam", DB_PARAM_OUTPUT, "",
                            10);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by the stored procedure. */
}
```

```
/* Close the statement.  Output parameters are invalid. */
/* until you close the statement.                       */
resCode = DBClosePreparedSQL(hstmt);
/* Examine the parameter values. */
resCode = DBGetParamChar(hstmt, 0, &retParam, "");
resCode = DBGetParamCharBuffer(hstmt, 1, readInParam, 10, "");
resCode = DBGetParamCharBuffer(hstmt, 2, outParam, 10, "");
printf("return param %s input param %s output param %s\n",
       retParam, inParam, outParam);
DBFree(retParam);
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBCreateParamChar, DBSetParamChar, DBGetParamChar, DBClosePreparedSQL

DBGetParamDouble

```
int status = DBGetParamDouble (int statementHandle, int index, double
                               *value);
```

Purpose

Obtains the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Output parameters are invalid until you close the statement with `DBClosePreparedSQL`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter. The index of the first parameter is `1`.

Output

Name	Type	Description
value	double-precision (passed by reference)	Double-precision value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamDouble` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam float, @OutParam float OUTPUT ) as  \
        select @OutParam = @InParam + 10.1 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam \
        +10.1")

/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");

/* Set command type back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN)

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamDouble(hstmt, 1, 10.5);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by the stored procedure. */
}

/* Close the statement.  Output parameters are invalid. */
/* until you close the statement.                       */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamDouble(hstmt, 0, &retParam);
resCode = DBGetParamDouble(hstmt, 1, &inParam);
resCode = DBGetParamDouble(hstmt, 2, &outParam);

/* Deactivate the statement. */
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamDouble, DBClosePreparedSQL

DBGetParamFloat

```
int status = DBGetParamFloat (int statementHandle, int index, float *value);
```

Purpose

Obtains the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Output parameters are invalid until you close the statement with `DBClosePreparedSQL`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter. The index of the first parameter is `1`.

Output

Name	Type	Description
value	float (passed by reference)	Floating-point value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamFloat` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam float, @OutParam float OUTPUT ) as  \
        select @OutParam = @InParam + 10.1 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam \
        +10.1");

/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");

/* Set command type back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamFloat(hstmt, 1, 10.5);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by the stored procedure. */
}

/* Close the statement.  Output parameters are invalid. */
/* until you close the statement.                       */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamFloat(hstmt, 0, &retParam);
resCode = DBGetParamFloat(hstmt, 1, &inParam);
resCode = DBGetParamFloat(hstmt, 2, &outParam);

/* Deactivate the statement. */
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBGetParamFloat, DBClosePreparedSQL

DBGetParamInt

```
int status = DBGetParamInt (int statementHandle, int index, int *value);
```

Purpose

Obtains the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Output values are invalid until you close the statement with `DBClosePreparedSQL`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter. The index of the first parameter is `1`.

Output

Name	Type	Description
value	integer (passed by reference)	Integer value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamInt` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam int, @OutParam int OUTPUT ) as  \
        select @OutParam = @InParam + 10 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam +10");

/* Prepare a statement that calls the stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamInt(hstmt, 1, 10);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* process records returned by the stored procedure. */
}

/* Close the statement.  Output parameters are invalid. */
/* until you close the statement.                       */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamInt(hstmt, 0, &retParam);
resCode = DBGetParamInt(hstmt, 1, &inParam);
resCode = DBGetParamInt(hstmt, 2, &outParam);

/* Deactivate the statement. */
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamInt, DBClosePreparedSQL

DBGetParamShort

```
int status = DBGetParamShort (int statementHandle, int index, short *value);
```

Purpose

Obtains the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Output parameters are invalid until you close the statement using `DBClosePreparedSQL`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter. The index of the first parameter is `1`.

Output

Name	Type	Description
value	short integer (passed by reference)	Short integer value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamShort` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam int, @OutParam int OUTPUT ) as  \
        select @OutParam = @InParam + 10 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam +10");

/* Prepare a statement that calls the stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamShort(hstmt, 1, 10);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* process records returned by the stored procedure */
}

/* Close the statement.  Output parameters are invalid. */
/* until you close the statement.                       */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamShort(hstmt, 0, &retParam);
resCode = DBGetParamShort(hstmt, 1, &inParam);
resCode = DBGetParamShort(hstmt, 2, &outParam);

/* Deactivate the statement. */
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamShort, DBClosePreparedSQL

DBGetParamVariant

```
int status = DBGetParamVariant (int statementHandle, int index,
                                VARIANT *value);
```

Purpose

Obtains the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Output parameters are invalid until you close the statement with `DBClosePreparedSQL`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter. The index of the first parameter is `1`.

Output

Name	Type	Description
value	`VARIANT` (passed by reference)	Variant that receives the value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamVariant` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam float, @OutParam float OUTPUT ) as  \
        select @OutParam = @InParam + 10.1 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam \
        +10.1");

/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");

/* Set command type back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamVariant(hstmt, 1, inParamV);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
...
/* Close the statement.  Output parameters are invalid.  */
/* until you close the statement.                        */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamVariant(hstmt, 0, &retParamV);
resCode = DBGetParamVariant(hstmt, 1, &inParamV);
resCode = DBGetParamVariant(hstmt, 2, &outParamV);

/* Deactivate the statement. */
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBSetParamDouble, DBClosePreparedSQL

DBGetSQLToolkitVersion

```
int versionNumber = DBGetSQLToolkitVersion (void);
```

Purpose

Obtains the version number of the LabWindows/CVI SQL Toolkit.

Return Value

Name	Type	Description
versionNumber	integer	Version number of the LabWindows/CVI SQL Toolkit. The version number is three decimal digits, so, for example, 200 stands for version 2.0.

DBGetStatementAttribute

```
int DBGetStatementAttribute (int statementHandle, tDBStatementAttr
                             attribute, void *value);
```

Purpose

Obtains a SQL statement attribute. You can obtain attributes for statements created with `DBActivateSQL`, `DBActivateMap`, `DBNewSQLStatement`, and `DBPrepareSQL`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
attribute	`tDBStatementAttr`	Attribute to get. Some providers do not support all attributes.

Output

Name	Type	Description
value	any type (passed by reference)	Value for the parameter attribute. The type of the value varies depending on the attribute. To free the strings that this function returns, use `DBFree`. To free allocated memory within the variants this function returns, use `CA_VariantClear`. Refer to *Parameter Discussion* for more information.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetStatementAttribute` returns. This function returns the set of result codes in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion for attribute and value Parameters

Attribute	Type	Description
ATTR_DB_STMT_PAGE_SIZE	long integer	Returns the number of records in a page.
ATTR_DB_STMT_ABSOLUTE_PAGE	long integer	Returns the current page. Page numbers start at one.
ATTR_DB_STMT_ABSOLUTE_POSITION	long integer	Returns the absolute position of the current record. Record numbers start at one.
ATTR_DB_STMT_CACHE_SIZE	long integer	Returns the number of records the provider keeps in its in memory buffer and the number of records the provider retrieves at one time.
ATTR_DB_STMT_CURSOR_TYPE	tDBCursorType	Returns one of the following cursor types: DB_CURSOR_TYPE_DYNAMIC—Additions, changes, and deletions by other users are visible, and all types of movement through the recordset are allowed. DB_CURSOR_TYPE_STATIC—Static copy of a set of records Additions, changes, or deletions by other users are not visible. DB_CURSOR_TYPE_FORWARD_ONLY—Identical to a static cursor except that you can only scroll forward through records. This setting improves performance when you only have to make a single pass through a recordset. DB_CURSOR_TYPE_KEYSET—Like a dynamic cursor, except that you cannot see records that other users add. Records that other users delete are inaccessible from your recordset. Data changes by other users within records continue to be visible.

Attribute	Type	Description
ATTR_DB_STMT_CURSOR_LOCATION	tDBCursorLoc	Returns one of the following cursor locations: DB_CURSOR_LOC_SERVER—Uses cursors that the data provider or driver provide. These cursors are sometimes very flexible and allow for some additional sensitivity to reflecting changes that others make to the actual data source. DB_CURSOR_LOC_CLIENT— Uses client-side cursors supplied by a local cursor library. Local cursor engines often allow many features that driver-supplied cursors do not.
ATTR_DB_STMT_LOCK_TYPE	tDBLockType	Returns one of the following lock types: DB_LOCK_READ_ONLY—You cannot alter the data. DB_LOCK_PESSIMISTIC—Provider ensures successful editing of the records, usually by locking records at the data source immediately when a user edits them. DB_LOCK_OPTIMISTIC—Provider locks records only when you call DBUpdateRecord. DB_LOCK_BATCH_OPTIMISTIC—Required for batch updates.
ATTR_DB_STMT_MAX_RECORDS	long integer	Returns the maximum number of records the provider returns from the data source. If 0, the provider returns all records.
ATTR_DB_STMT_MARSHAL_OPTIONS	tDBMarshalOpt	Returns one of the following modes for writing data back to the server: DB_MARSHAL_OPT_ALL—All records are written back to the server. DB_MARSHAL_OPT_MODIF_ONLY—Only modified data is written back to the server.
ATTR_DB_STMT_BOOKMARK	Variant	Returns a bookmark for the current record.

Attribute	Type	Description
ATTR_DB_STMT_COMMAND_TYPE	tDBCommandType	Returns one of the following values regarding interpretation of the input text: DB_COMMAND_UNKNOWN—ADO cannot determine the command type. DB_COMMAND_TEXT—SQL statement or command in provider-specific language. DB_COMMAND_TABLE—Table name. DB_COMMAND_STORED_PROC—Call to a stored procedure.
ATTR_DB_STMT_COMMAND_TIMEOUT	long integer	Time in seconds to wait for a command to execute. If 0, wait for an unlimited time period.
ATTR_DB_STMT_PREPARED	long integer	Whether to save a prepared (compiled) version of the statement to speed future executions of the statement.
ATTR_DB_STMT_ACTIVE_CONNECTION	String	String defining a connection. The same as the connection string used for DBConnect.
ATTR_DB_STMT_NAME	String	Name of the command. Applies only to statements created with DBPrepareSQL.

Attribute	Type	Description
ATTR_DB_STMT_FILTER	Variant	Returns the current filter, one of the following three: - A criteria string made up of individual clauses connected by AND or OR. - An array of bookmarks. - A filter group with one of the following values: DB_FILTER_NONE—Removes the current filter. DB_FILTER_PENDING—Only records that have changed but have not yet been sent to the server. Only applicable for batch update mode. DB_FILTER_AFFECTED—Only records affected by the last DBDeleteRecord or DBUpdateBatch. DB_FILTER_FETCHED—Records in the current cache.
ATTR_DB_STMT_PAGE_COUNT	long integer	Returns the number of pages in the current recordset. If -1, the provider does not support pages.
ATTR_DB_STMT_RECORD_COUNT	long integer	Returns the number of records in the current recordset. If -1, the provider cannot determine the number of records.
ATTR_DB_STMT_BOF	long integer	Returns whether the current record pointer is located before the beginning of the recordset.
ATTR_DB_STMT_EOF	long integer	Returns whether the current record pointer is located after the end of the recordset.

Attribute	Type	Description
ATTR_DB_STMT_EDIT_MODE	tDBEditMode	Returns one of the following edit modes for the current record: DB_EDIT_MODE_NONE—No edit in progress. DB_EDIT_MODE_IN_PROGRESS—Current record has been changed. DB_EDIT_MODE_ADD—Current record is a new record. DB_EDIT_MODE_DELETE—Current record has been deleted.
ATTR_DB_STMT_STATUS	long integer	Returns the status of the current record with respect to batch updates or other bulk operations. The status is the sum or one or more of the following tDBRecStatus values: DB_REC_STATUS_OK—Record was successfully updated DB_REC_STATUS_NEW—Record is new. DB_REC_STATUS_MODIFIED—Record was modified. DB_REC_STATUS_DELETED—Record was deleted. DB_REC_STATUS_UNMODIFIED—Record was unmodified. DB_REC_STATUS_INVALID—Record was not saved because its bookmark is invalid. DB_REC_STATUS_MULTIPLE_CHANGES—Record was not saved because it would have affected multiple records. DB_REC_STATUS_PENDING_CHANGES—Record was not saved because it refers to a pending insert. DB_REC_STATUS_CANCELED—Record was not saved because the operation was cancelled. *continues*

Attribute	Type	Description
continued ATTR_DB_STMT_STATUS	long integer	DB_REC_STATUS_CANT_RELEASE—Record was not saved because of existing record locks. DB_REC_STATUS_CONCURRENCY_VIOLATION—Record was not saved because optimistic concurrency was in use. DB_REC_STATUS_INTEGRITY_VIOLATION—Record was not saved because the user violated integrity constraints. DB_REC_STATUS_MAX_CHANGES_EXCEEDED—Record was not saved because there were too many pending changes. DB_REC_STATUS_OBJECT_OPEN—Record was not saved because of a conflict with an open storage object. DB_REC_STATUS_OUT_OF_MEMORY—Record was not saved because the computer has run out of memory. DB_REC_STATUS_PERMISSION_DENIED—Record was not saved because the user has insufficient permissions. DB_REC_STATUS_SCHEMA_VIOLATION—Record was not saved because it violates the structure of the underlying database. DB_REC_STATUS_DBDELETED—Record has already been deleted from the data source.
ATTR_DB_STMT_STATE	tDBObjectState	Returns whether the statement is open.
ATTR_DB_STMT_RECORDSET_OBJECT	CAObjHandle	ActiveX object handle of the recordset when the statement uses a recordset; otherwise, the handle is zero (0).
ATTR_DB_STMT_COMMAND_OBJECT	CAObjHandle	ActiveX object handle of the command when the statement uses a command; otherwise, the handle is zero (0).

Parameter Discussion

Do not use `DBFree` to free items that you extract from variants. To free allocated memory within the variants that `DBGetStatementAttribute` returns, use `CA_VariantClear`. For information on functions you can use to free resources, refer to Chapter 11, *ActiveX Automation Library*, of the *LabWindows/CVI Standard Libraries Reference Manual*.

Examples

```
hstmt = DBNewSQLStatement (hdbc, "SELECT UUT_NUM, MEAS1, \
                            MEAS2 FROM TESTRES");
/* Set the cursor type. */
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_CURSOR_TYPE,
                                   DB_CURSOR_TYPE_FORWARD_ONLY);
...
resCode = DBOpenSQLStatement(hstmt);
/* Set the absolute position. */
resCode = DBSetStatementAttribute(hstmt,
                                  ATTR_DB_STMT_ABSOLUTE_POSITION, 2);
...
resCode = DBCloseSQLStatement(hstmt);
```

Bookmark and Filter Example

```
VARIANT vFilterArray;
int filterIndex = 0;
VARIANT bookmarks[2];

hstmt = DBNewSQLStatement (hdbc, "SELECT UUT_NUM, MEAS1, \
                                   MEAS2 FROM TESTRES");
resCode = DBOpenSQLStatement(hstmt);

/* Criteria string example.                              */
/* Note: It almost always much more efficient to use a   */
/* where clause in the SQL statement, instead of using a */
/* criteria string in a filter.                          */
CA_VariantSetCString (&filter, "(MEAS1 > 1.0)");
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_FILTER,
                                   filter);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* only records where MEAS1 greater than 1.0. */
    ...
}

/* Filter constant example. */
CA_VariantSetLong (&filter, DB_FILTER_NONE);
resCode = DBSetStatementAttribute(hstmt,
        ATTR_DB_STMT_FILTER, filter);
```

```
i = 0;
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Will get all records. */
    ...
    /* Get some bookmarks to use in next example. */
    if ((i == 0) || (i == 2)) {
        resCode = DBGetStatementAttribute(hstmt,
                ATTR_DB_STMT_BOOKMARK,
                &(bookmarks[filterIndex++]));
    i++;
    ...
}
/* Bookmark array example. */
CA_VariantSet1DArray (&vFilterArray, CAVT_VARIANT, 2, bookmarks);
resCode = DBSetStatementAttribute(hstmt,
    ATTR_DB_STMT_FILTER, vFilterArray);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Will get only records at the bookmarks we   */
    /* obtained in the previous example.           */
    ...
}
resCode = DBCloseSQLStatement(hstmt);
```

DBGetVariantArray

```
int status = DBGetVariantArray (int statementHandle,
                                VARIANT **returnedArray, unsigned int
                                *recordsReturned, unsigned int *fieldsReturned);
```

Purpose

Return the records for the current statement as an array of `VARIANT` pointers. `DBGetVariantArray` is faster, but more complicated to use than the `DBFetch` functions or the `DBGetCol` functions.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
returnedArray	`VARIANT *` (passed by reference)	Array of variants that contains the values of the records and fields you requested. When you no longer need the array, use `DBFreeVariantArray` to clear all the variants of the array and free the array.
recordsReturned	unsigned integer (passed by reference)	Number of records/rows returned in the array.
fieldsReturned	unsigned integer (passed by reference)	Number of fields/columns returned in the array.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetVariantArray` returns. Result codes are the same as those that `DBError` returns.

Example

```
/* Execute a select statement. */
VARIANT *cArray;
...
hstmt = DBActivateSQL (hdbc, "SELECT * FROM TESTRES");
resCode = DBGetVariantArray(hstmt, &cArray,
                            &numRecs, &numFields);
for (i = 0; i < numRecs; i++) {
    for (j = 0; j < numFields; j++) {
        resCode =
            DBGetVariantArrayValue(cArray, numRecs, numFields,
                                   CAVT_CSTRING, i, j, &tempStr);
        if (resCode == DB_NULL_DATA){
            /* Handle null data.  */
        }else {
            /* Handle other data. */
            DBFree(tempStr);
        }
    }
}
resCode = DBFreeVariantArray(cArray, 1, numRecs, numFields);
```

See Also

DBGetVariantArrayValue, DBGetVariantArrayColumn, DBFreeVariantArray

DBGetVariantArrayColumn

```
int status = DBGetVariantArrayColumn (VARIANT *recordsArray, unsigned int
                            recordsReturned, unsigned int fieldsReturned,
                            unsigned int desiredType, unsigned int
                            fieldNumber, unsigned int firstRecordNumber,
                            unsigned int numberofRecordstoGet, void *value);
```

Purpose

Returns the value at the specified record and field numbers from the array that `DBGetVariantArray` returns and converts the value to the specified type.

Note ***If any record contains a SQL Null value in the specified field,*** `DBGetVariantArrayColumn` ***stops processing records and returns*** `DB_NULL_DATA` ***immediately.***

Parameters

Input

Name	Type	Description
recordsArray	`VARIANT *`	Array of variants that contains the values of the records and fields you requested. When you no longer need the array, use `DBFreeVariantArray` to clear all the variants of the array and free the array.
recordsReturned	unsigned integer	Number of records/rows in the array. Use the records returned value from `DBGetVariantArray` for this value.
fieldsReturned	unsigned integer	Number of fields/columns in the array. Use the fields returned value from `DBGetVariantArray` for this value.
desiredType	unsigned integer	Desired type of the returned value. Refer to *Parameter Discussion.*
fieldNumber	unsigned integer	Number of the field/column to get as a sub-array. The first field is `0`.
firstRecordNumber	unsigned integer	Number of the first record/row to get. The first record in the array is number `0`.
numberofRecordstoGet	unsigned integer	Number of records to get.

Output

Name	Type	Description
value	void * (passed by reference)	Array of field values for the specified field converted to the specified type. Pass the address of an array that is large enough to hold the converted values. If the type of the converted values is `char *` or `BSTR`, you must call the appropriate function to free the values when they are no longer needed. For `char *` call `DBFree`. For `BSTR` call `SysFreeString` (a Windows SDK function).

Return Value

Name	Type	Description
status	integer	Result code that `DBGetVariantArrayColumn` returns. Result codes are the same as those that `DBError` returns. If any of the records contain `NULL` in the specified field, the function stops processing records and returns `DB_NULL_DATA`.

Parameter Discussion

The following table lists type constants and corresponding value types:

Type Constant	Value Type
CAVT_LONG	long
CAVT_SHORT	short
CAVT_INT	int
CAVT_BOOL	VBOOL
CAVT_FLOAT	float
CAVT_DOUBLE	double
CAVT_CY	CURRENCY
CAVT_DATE	DATE
CAVT_CSTRING	char*
CAVT_BSTR	BSTR
CAVT_VARIANT	VARIANT
CAVT_ERROR	SCODE
CAVT_UCHAR	unsigned char

Example

```
/* Execute a select statement */
double *column;
...
hstmt = DBActivateSQL (hdbc, "SELECT * FROM TESTRES");

resCode = DBGetVariantArray(hstmt, &cArray, &numRecs, &numFields);
column = malloc(numRecs * sizeof(double));

resCode = DBGetVariantArrayColumn(cArray, numRecs, numFields,
                            CAVT_DOUBLE, 1, 2, 3, column);
if (resCode == DB_NULL_DATA) {
    printf("Cannot process, some fields contain null\n");
} else {
    for(i = 0; i < 3; i++) {
       /* Process values.                                  */
       /* Note: Because the type of the values is not    */
       /* char* or BSTR, the values should not be freed. */
```

```
    }
}
resCode = DBFreeVariantArray(cArray, 1, numRecs, numFields);
free(column);
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBGetVariantArray, DBGetVariantArrayColumn, DBFreeVariantArray

DBGetVariantArrayValue

```
int status = DBGetVariantArrayValue (VARIANT *recordsArray, unsigned int
                            recordsinArray, unsigned int fieldsinArray,
                            unsigned int desiredType, unsigned int
                            recordNumber, unsigned int fieldNumber, void
                            *value);
```

Purpose

Returns the value at the specified record and field numbers from the array that `DBGetVariantArray` returns and converts the value to the specified type.

Parameters

Input

Name	Type	Description
recordsArray	`VARIANT *`	Array of variants that contains the values of the records and fields you requested. When you no longer need the array, use `DBFreeVariantArray` to clear all the variants of the array and free the array.
recordsinArray	unsigned integer	Number of records/rows in the array. Use the records returned value from `DBGetVariantArray` for this value.
fieldsinArray	unsigned integer	Number of fields/columns in the array. Use the Fields returned value from `DBGetVariantArray` for this value.
desiredType	unsigned integer	Desired type of the returned value. Refer to the *Parameter Discussion*.
recordNumber	unsigned integer	Number of the record/row of the value. The first record is `0`.
fieldNumber	unsigned integer	Number of the field/column of the value. The first field is `0`.

Output

Name	Type	Description
value	void * (passed by reference)	Value at the specified record and field numbers converted to the requested type. Pass the address of a variable large enough to hold the converted value. If the type of the converted value is `char *` or `BSTR`, you must call the appropriate function to free the value when it is no longer needed. For `char *` call `DBFree`. For `BSTR` call `SysFreeString` (a Windows SDK function).

Return Value

Name	Type	Description
status	integer	Result code that `DBGetVariantArrayValue` returns. Result codes are the same as those that `DBError` returns.

Parameter Discussion

The following table lists type constants and corresponding value types:

Type Constant	Value Type
`CAVT_LONG`	`long`
`CAVT_SHORT`	`short`
`CAVT_INT`	`int`
`CAVT_BOOL`	`VBOOL`
`CAVT_FLOAT`	`float`
`CAVT_DOUBLE`	`double`
`CAVT_CY`	`CURRENCY`
`CAVT_DATE`	`DATE`

Type Constant	Value Type
CAVT_CSTRING	char*
CAVT_BSTR	BSTR
CAVT_VARIANT	VARIANT
CAVT_ERROR	SCODE
CAVT_UCHAR	unsigned char

Example

```
/* Execute a select statement. */
VARIANT *cArray;
...
hstmt = DBActivateSQL (hdbc, "SELECT * FROM TESTRES");

resCode = DBGetVariantArray(hstmt, &cArray, &numRecs, &numFields);
for (i = 0; i < numRecs; i++) {
    for (j = 0; j < numFields; j++) {
        resCode =
            DBGetVariantArrayValue(cArray, numRecs, numFields,
                                   CAVT_CSTRING, i, j, &tempStr);
        if (resCode == DB_NULL_DATA){
            /* Handle null data. */
        }else{
            /* Handle other data. */
            DBFree(tempStr);
        }
    }
}
resCode = DBFreeVariantArray(cArray, 1, numRecs, numFields);
```

See Also

DBGetVariantArray, DBGetVariantArrayColumn, DBFreeVariantArray

DBImmediateSQL

```
int status = DBImmediateSQL (int connectionHandle, char SQLStatement[]);
```

Purpose

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

If you execute a `SELECT` statement with `DBImmediateSQL`, you cannot access the selected records, and some database systems do not release the locks for tables in the `SELECT` statement.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returned.
SQLStatement	char []	SQL statement the function executes.

Return Value

Name	Type	Description
status	integer	Result code that `DBImmediateSQL` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hdbc = DBConnect("DSN=CVI32_Samples");
...
resCode = DBImmediateSQL (hdbc,
                          "CREATE TABLE TESTRES \
                          (UUT_NUM CHAR (10), \
                          MEAS1 NUMERIC (10,2), \
                          MEAS2 NUMERIC (10,2))");
resCode = DBImmediateSQL (hdbc,
        "INSERT INTO TESTRES VALUES ('28A123', 0.5 ,0.5)");
resCode = DBImmediateSQL (hdbc,
        "INSERT INTO TESTRES VALUES ('28A124', 0.0 ,0.5)");
...
resCode = DBDisconnect(hdbc);
```

See Also

DBNumberOfModifiedRecords, DBActivateSQL, DBDeactivateSQL

DBIndexes

```
int status = DBIndexes (int connectionHandle, char tableName[], short flags);
```

Purpose

Creates and activates a `SELECT` statement that returns information about the set of indexes on a table. You can then use the `DBFetch` and `DBBindCol` or `DBGetCol` functions to retrieve the information. Each record contains the following columns:

Column	Type	Description
Table Catalog	string	Table catalog. This is a path for file-based databases.
Table Schema	string	Table schema.
Table Name	string	Table name (table-based source). File name (file-based source).
Index Catalog	string	Index catalog.
Index Schema	string	Index schema.
Index Name	string	Index name.
Unique	short integer	Indicates whether or not every index entry must be unique. TRUE or `1` = index values must be unique. FALSE or `0` = index value do not have to be unique.
Clustered	integer	Clustered.
Type	string	Index type.
Initial Size	string	Initial size.
Allow NULLs	short integer	`NULL` is allowed.
Sort Bookmark	integer	Sort bookmark.
Auto Update	integer	Column is auto update.
NULL Collation	integer	`NULL` collation.
Sequence Number	short integer	Number of this column within the index.
Column Name	string	Column name.

Column	Type	Description
Column GUID	integer	Column GUID.
Column Property ID	integer	Column property ID.

Collation	char (1)	Collating sequence. Values:
		`DB_INDEX_ASC`—`'A'`
		`DB_INDEX_DESC`—`'D'`
		`DB_INDEX_ORDER_UNKNOWN`—`NULL`
		This value is a string, so you must reserve space for the NUL string termination character.

Cardinality	integer	Number of unique values in index; can be `NULL`.
Pages	integer	Number of pages used to store index; can be `NULL`.
Filter	string	Filter condition when one exists, otherwise `NULL`.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returned.
tableName	char []	Pointer to a string containing the name of the table for which index information is selected.
flags	short integer	Specifies a set of option flags that control the values returned from `DBIndexes`.

Return Value

Name	Type	Description
statementHandle	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Example

```
hstmt = DBIndexes(hdbc, "testres", 0x0);

resCode = DBBindColChar (hstmt, 1, 128, tableCatalog, &stat1, "");
resCode = DBBindColChar (hstmt, 2, 128, tableSchema, &stat2, "");
...
resCode = DBBindColChar (hstmt, 21, 2, collation, &stat10, "");
...
resCode = DBBindColChar (hstmt, 24, 128, filter, &stat13, "");

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
...
}
resCode = DBDeactivateSQL(hstmt);
```

DBInit

```
int status = int DBInit (int options);
```

Purpose

Initialize the SQL toolkit. If you use the toolkit functions from multiple threads, you must initialize the toolkit before calling any toolkit functions. You do not need to initialize the toolkit if you only use the toolkit from a single thread.

Note ***You cannot initialize the toolkit for use from multiple threads if you use the Microsoft ODBC drivers for Access, DBase, FoxPro, Paradox, Excel, or text files.***

Parameter

Input

Name	Type	Description
options	integer	Initialization options. `DB_INIT_SINGLE_THREADED`—Initialize for single threaded (default). `DB_INIT_MULTI_THREADED`—Initialize for multithreaded.

Return Value

Name	Type	Description
status	integer	Result code returned by `DBInit`. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to get the text of the error message.

Example

```
resCode = DBInit(DB_INIT_MULTITHREADED);
hdbc = DBConnect("DSN=CVI Samples");
```

DBMapColumnToBinary

```
int status = DBMapColumnToBinary (int mapHandle, char columnName[], unsigned
                                  long maximumLength, void *locationforValue, long
                                  *locationforStatus);
```

Purpose

Specifies a column to select and the value and status variables in your program that receive the value and status of a column each time a `DBFetch` function fetches a record.

Parameters

Input

Name	Type	Description
mapHandle	integer	Handle to the map that `DBBeginMap` returned.
columnName	char []	Variables of the column name that the function specifies. If you connect to an existing table, but use a column name that is not in the table, some database systems interpret the invalid name as a parameter. The resulting error message could be misleading. Some database systems have restrictions on column names. For maximum portability, limit column names to ten uppercase characters with no space characters, or enclose the column name in the ASCII grave character (`).
maximumLength	unsigned long integer	Size of the binary data in bytes.
locationforValue	void pointer (passed by reference)	Pointer to the variable that receives the binary value for the column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record.

Return Value

Name	Type	Description
status	integer	Result code that `DBMapColumnToChar` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values:

Value Name	Value	Description
`DB_TRUNCATION`	`-1`	Data is truncated.
`DB_NULL_DATA`	`-2`	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
unsigned char *toDBBits = NULL;
long bitsStatus = 0;
int bitsSize = 6;
...
toDBBits = malloc(bitsSize);
toDBBits[0] = 'N'; toDBBits[1] = 0; toDBBits[2] = 'C';
toDBBits[3] = 'B'; toDBBits[4] = 0; toDBBits[5] = 250;

/* Save the data. */
map = DBBeginMap (hdbc);

resCode = DBMapColumnToBinary(map, "THE BITS", bitsSize, toDBBits,
                              &bitsStatus);

resCode = DBCreateTableFromMap (map, "BINTEST");

hstmt = DBActivateMap (map, "BINTEST");
resCode = DBCreateRecord(hstmt);
resCode = DBPutRecord(hstmt);
```

```
resCode = DBDeactivateMap(map);
hstmt = 0;
map = 0;
```

See Also

DBBeginMap, DBActivateMap, DBFetchNext, DBFetchPrev, DBFetchRandom, DBPutRecord, DBDeactivateMap

DBMapColumnToChar

```
int status = DBMapColumnToChar (int mapHandle, char columnName[], unsigned
                                long maximumLength, char locationforValue[], long
                                *locationforStatus, char formatString[]);
```

Purpose

Specifies a column to select and the value and status variables in your program that receive the value and status of a column each time a `DBFetch` function fetches a record.

Parameters

Input

Name	Type	Description
mapHandle	integer	Handle to the map that `DBBeginMap` returned.
columnName	char []	Variables of the column name that the function specifies. If you connect to an existing table, but use a column name that is not in the table, some database systems interpret the invalid name as a parameter. The resulting error message could be misleading. Some database systems have restrictions on column names. For maximum portability, limit column names to ten uppercase characters with no space characters, or enclose the column name in the ASCII grave character (`).
maximumLength	long integer	Size of the target string in bytes. The toolkit uses one byte of the target string for the string termination character, NUL.
locationforValue	char []	Pointer to the variable that receives the null-terminated character string value for the column when you fetch a record.

Name	Type	Description
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record. See *Parameter Discussion*.
formatString	char []	Format string. Use the empty string, `""`, if you want the default format. See Appendix C, *Format Strings*, for details about formatting.

Return Value

Name	Type	Description
status	integer	Result code that `DBMapColumnToChar` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values for **locationforStatus**:

Value Name	Value	Description
`DB_TRUNCATION`	`-1`	Data is truncated.
`DB_NULL_DATA`	`-2`	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
char serialNum[11];
long sNumStatus;
...
hmap = DBBeginMap(hdbc);
resCode = DBMapColumnToChar(hmap, "SER_NUM", 11, serialNum,
                            &sNumStatus, "");
/* More variable mappings. */
...
hstmt = DBActivateMap(map, "testlog");
while (DBFetchNext(hstmt) == 0) {
    if (sNumStatus == DB_NULL_DATA)
       ...
    if (sNumStatus == DB_TRUNCATION)
       ...
    printf("Serial Number: %s\n",serialNumber);
    ...
}
resCode = DBDeactivateMap(hmap);
```

See Also

DBBeginMap, DBActivateMap, DBFetchNext, DBFetchPrev, DBFetchRandom, DBPutRecord, DBDeactivateMap

DBMapColumnToDouble

```
int status = DBMapColumnToDouble (int mapHandle, char columnName[], double
                                  *locationforValue, long *locationforStatus);
```

Purpose

Specifies a column to select and the value and status variables in your program that receive the value and status of a column each time a `DBFetch` function fetches a record.

Parameters

Input

Name	Type	Description
mapHandle	integer	Handle to the map that `DBBeginMap` returned.
columnName	char []	Variables of the column name that the function specifies. If you connect to an existing table, but use a column name that is not in the table, some database systems interpret the invalid name as a parameter. The resulting error message could be misleading. Some database systems have restrictions on column names. For maximum portability, limit column names to ten uppercase characters with no space characters, or enclose the column name in the ASCII grave character (`).
locationforValue	double-precision (passed by reference)	Pointer to the variable that receives the double-precision value for the column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record. See *Parameter Discussion*.

Return Value

Name	Type	Description
status	integer	Result code that `DBMapColumnToDouble` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values for **locationforStatus**:

Value Name	Value	Description
`DB_TRUNCATION`	`-1`	Data is truncated.
`DB_NULL_DATA`	`-2`	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
float measurement;
long measStatus;
...
hmap = DBBeginMap(hdbc);
resCode = DBMapColumnToDouble(hmap, "MEAS1", &measurement,
                              &measStatus);
/* More variable mappings. */
...
hstmt = DBActivateMap(hmap, "TESTLOG");
```

```
while (DBFetchNext(hstmt) == 0) {
    if (measStatus == DB_NULL_DATA)
       ...
    if (measStatus == DB_TRUNCATION)
       ...
    printf("Measurement: %f\n",measurement);
    ...
}
resCode = DBDeactivateMap(hmap);
```

See Also

DBBeginMap, DBActivateMap, DBFetchNext, DBFetchPrev, DBFetchRandom, DBDeactivateMap, DBPutRecord

DBMapColumnToFloat

```
int status = DBMapColumnToFloat (int mapHandle, char columnName[], float
                                 *locationforValue, long *locationforStatus);
```

Purpose

Specifies a column to select and the value and status variables in your program that receive the value and status of a column each time a `DBFetch` function fetches a record.

Parameters

Input

Name	Type	Description
mapHandle	integer	Handle to the map that `DBBeginMap` returned.
columnName	char []	Variables of the column name that the function specifies. If you connect to an existing table, but use a column name that is not in the table, some database systems interpret the invalid name as a parameter. The resulting error message could be misleading. Some database systems have restrictions on column names. For maximum portability, limit column names to ten uppercase characters with no space characters, or enclose the column name in the ASCII grave character (`).
locationforValue	float (passed by reference)	Pointer to the variable that receives the floating-point value for the column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record. See *Parameter Discussion*.

Return Value

Name	Type	Description
status	integer	Result code that `DBMapColumnToFloat` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values for **locationforStatus**:

Value Name	Value	Description
`DB_TRUNCATION`	`-1`	Data is truncated.
`DB_NULL_DATA`	`-2`	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
float measurement;
long measStatus;
...
hmap = DBBeginMap(hdbc);
resCode = DBMapColumnToFloat(hmap, "measurement", &measurement,
                             &measStatus);
/* More variable mappings. */
...
hstmt = DBActivateMap(hmap, "testlog");
while (DBFetchNext(hstmt) == 0) {
    if (measStatus == DB_NULL_DATA)
       ...
    if (measStatus == DB_TRUNCATION)
       ...
    printf("Measurement %f\n",measurement);
    ...
}
resCode = DBDeactivateMap(hmap);
```

See Also

DBBeginMap, DBActivateMap, DBFetchNext, DBFetchPrev, DBFetchRandom, DBDeactivateMap, DBPutRecord

DBMapColumnToInt

```
int status = DBMapColumnToInt (int mapHandle, char columnName[],
                               int *locationforValue, long *locationforStatus);
```

Purpose

Specifies a column to select and the value and status variables in your program that receive the value and status of a column each time a `DBFetch` function fetches a record.

Parameters

Input

Name	Type	Description
mapHandle	integer	Handle to the map that `DBBeginMap` returns.
columnName	char []	Variables of the column name that the function specifies. If you connect to an existing table, but use a column name that is not in the table, some database systems interpret the invalid name as a parameter. The resulting error message could be misleading. Some database systems have restrictions on column names. For maximum portability, limit column names to ten uppercase characters with no space characters, or enclose the column name in the ASCII grave character (`).
locationforValue	integer (passed by reference)	Pointer to the variable that receives the integer value for the column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record. See *Parameter Discussion*.

Return Value

Name	Type	Description
status	integer	Result code that `DBMapColumnToInt` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values:

Value Name	Value	Description
`DB_TRUNCATION`	`-1`	Data is truncated.
`DB_NULL_DATA`	`-2`	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
short numTries;
long numTriesStatus;
...
hmap = DBBeginMap(hdbc);
resCode = DBMapColumnToInt(hmap, "num_tries", &numTries,
                           &numTriesStatus);
/* More variable mappings. */
...
hstmt = DBActivateMap(hmap, "testlog");
```

```
while (DBFetchNext(hstmt) == 0) {
    if (sNumStatus == DB_NULL_DATA)
       ...
    if (sNumStatus == DB_TRUNCATION)
       ...
    printf("Number of tries: %d\n",numTries);
    ...
}
resCode = DBDeactivateMap(hmap);
```

See Also

DBBeginMap, DBActivateMap, DBFetchNext, DBFetchPrev, DBFetchRandom, DBDeactivateMap, DBPutRecord

DBMapColumnToShort

```
int status = DBMapColumnToShort (int mapHandle, char columnName[], short
                          *locationforValue, long *locationforStatus);
```

Purpose

Specifies a column to select and the value and status variables in your program that receive the value and status of a column each time a `DBFetch` function fetches a record.

Parameters

Input

Name	Type	Description
mapHandle	integer	Handle to the map that `DBBeginMap` returned.
columnName	char []	Variables of the column name that the function specifies. If you connect to an existing table, but use a column name that is not in the table, some database systems interpret the invalid name as a parameter. The resulting error message could be misleading. Some database systems have restrictions on column names. For maximum portability, limit column names to ten uppercase characters with no space characters, or enclose the column name in the ASCII grave character (`).
locationforValue	short integer (passed by reference)	Pointer to the variable that receives the short integer value for the column when you fetch a record.
locationforStatus	long integer (passed by reference)	Pointer to the variable that receives the status value for the column when you fetch a record. See *Parameter Discussion*.

Return Value

Name	Type	Description
status	integer	Result code that `DBMapColumnToShort` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion

The following table shows the possible status values:

Value Name	Value	Description
`DB_TRUNCATION`	`-1`	Data is truncated.
`DB_NULL_DATA`	`-2`	Null data.
(none)	positive integer	Number of bytes fetched.

Note ***You can use*** `DB_NULL_DATA` ***to place a NULL value into a column as follows: set the status value for that column to*** `DB_NULL_DATA`***, then call*** `DBPutRecord`***. To prevent*** `DBPutRecord` ***from updating a column, set the status value to*** `DB_NO_DATA_CHANGE`***.*** `DB_NO_DATA_CHANGE` ***is useful when the record contains read-only columns.***

Example

```
short numTries;
long numTriesStatus;
...
hmap = DBBeginMap(hdbc);
resCode = DBMapColumnToShort(hmap, "NUM_TRIES", &numTries,
                             &numTriesStatus);
/* More variable mappings. */
...
hstmt = DBActivateMap(hmap, "TESTLOG");
while (DBFetchNext(hstmt) == 0) {
    if (sNumStatus == DB_NULL_DATA)
       ...
    if (sNumStatus == DB_TRUNCATION)
       ...
    printf("Number of tries: %d\n",numTries);
    ...
}
resCode = DBDeactivateMap(hmap);
```

See Also

DBBeginMap, DBActivateMap, DBFetchNext, DBFetchPrev, DBFetchRandom, DBDeactivateMap, DBPutRecord

DBMoreResults

```
int status = DBMoreResults (int statementHandle);
```

Purpose

Moves to the next set of results from a compound SQL statement, for example, `"SELECT * FROM table1;SELECT * FROM table2"`. `DBMoreResults` removes all existing bindings; after you use this function you must establish bindings again.

Most providers support compound statements only in stored procedures.

Note ***Because mapping can never produce multiple sets of records,*** `DBMoreResults` ***does not apply to the mapping functions.***

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBNewSQLStatement`, `DBPrepareSQL`, or `DBActivateSQL` returns.

Return Value

Name	Type	Description
status	integer	Result code that `DBMoreResults` returns. `DBMoreResults` returns the set of result codes listed in the function description for `DBError`.

Example

```
resCode = DBImmediateSQL(hdbc, "create proc sp_moreres \
                        as SELECT * FROM TESTRES \
                        SELECT * FROM MORERES");

hstmt = DBNewSQLStatement (hdbc, "sp_moreres");
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_COMMAND_TYPE,
                                  DB_COMMAND_STORED_PROC);
resCode = DBOpenSQLStatement(hstmt);
/* bind the columns */
DBBindColChar(hstmt, 1, serialNumLen, serialNum, &serialNumStat, "");
...
```

```
/* first set of values */
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
resCode = DBMoreResults(hstmt);
/* Bind the columns in the second record set, which may be  */
/* different from those in the first record set.            */
DBBindColInt(hstmt, 5, &numTries, &numTriesStat);
...
/* second set of values */
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
DBCloseSQLStatement(hstmt);
DBDiscardSQLStatement(hstmt);
```

DBNativeError

```
int status = DBNativeError (void);
```

Purpose

Returns the result code from the underlying database system for the last SQL Toolkit Library function you called. You can call `DBNativeError` when `DBError` returns a value of `DB_DBSYS_ERROR` or `DB_AUTOMATION_ERROR` to determine the native error code.

Return Value

Name	Type	Description
status	integer	Native error code in the underlying database system. If zero, no database system error was reported. If `DBError` returned `DB_DBSYS_ERROR` or `DB_AUTOMATION_ERROR`, `DBNativeError` might return `0`. In this case, the underlying database system does not have a separate error code.

See Also

`DBError`, `DBWarning`, `DBErrorMessage`

DBNewConnection

```
int connectionHandle = DBNewConnection (void);
```

Purpose

Creates an unopened connection. Unlike `DBConnect`, `DBNewConnection`, in conjunction with `DBOpenConnection`, allows you to set connection attributes before opening the connection.

Return Value

Name	Type	Description
connectionHandle	integer	Database connection handle that identifies the connection and is a parameter to other functions. If the handle is less than or equal to `0`, the connection could not be opened.

Note ***You must use*** `DBInit` ***before*** `DBConnect` ***when your program is multithreaded.***

Example

```
hdbc = DBNewConnection();
resCode = DBSetConnectionAttribute(hdbc,
                                   ATTR_DB_CONN_CONNECTION_TIMEOUT, 100);
resCode = DBSetConnectionAttribute(hdbc,
                                        ATTR_DB_CONN_ISOLATION_LEVEL,
                                        DB_ISOLATION_LEVEL_SERIALIZABLE);
resCode = DBOpenConnection(hdbc);
...
resCode = DBCloseConnection(hdbc);
resCode = DBDiscardConnection(hdbc);
```

See Also

`DBConnect`, `DBOpenConnection`, `DBSetConnectionAttribute`, `DBGetConnectionAttribute`, `DBCloseConnection`, `DBDiscardConnection`

DBNewSQLStatement

```
int statementHandle = DBNewSQLStatement (int connectionHandle, char
                          SQLStatement[]);
```

Purpose

Creates, but does not open or execute, a SQL statement. `DBNewSQLStatement` (in conjunction with `DBOpenSQLStatement`) allows you to set attributes of the statement before executing the statement.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returned.
SQLStatement	char []	Stored procedure or SQL statement to activate.

Return Value

Name	Type	Description
statementHandle	integer	Statement handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

☞ **Note** ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Example

```
hstmt = DBNewSQLStatement (hdbc, "SELECT UUT_NUM, MEAS1,\
                                  MEAS2 FROM TESTRES");
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_MAX_RECORDS,
                                  1);
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_CACHE_SIZE,
                                  10);
resCode = DBOpenSQLStatement(hstmt);
...
resCode = DBGetStatementAttribute(hstmt, ATTR_DB_STMT_RECORD_COUNT,
                                  &recordCount
...
resCode = DBCloseSQLStatement(hstmt);
resCode = DBDiscardSQLStatement(hstmt);
```

See Also

DBOpenSQLStatement, DBCloseSQLStatement, DBDiscardSQLStatement, DBSetStatementAttribute, DBGetStatementAttribute, DBActivateSQL, DBPrepareSQL

DBNumberOfColumns

```
int numberofColumns = DBNumberOfColumns (int statementHandle);
```

Purpose

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

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBActivateSQL`, `DBActivateMap`, or `DBPrepareSQL` returned.

Return Value

Name	Type	Description
numberofColumns	integer	Number of columns selected. If zero, the statement is not a `SELECT` statement. If less than zero, an error occurred.

Example

```
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
numCols = DBNumberOfColumns(hstmt);
...
resCode = DBDeactivateSQL(hstmt);
```

See Also

`DBActivateSQL`, `DBDeactivateMap`

DBNumberOfModifiedRecords

```
int numberofRecords = DBNumberOfModifiedRecords (int statementHandle);
```

Purpose

Returns the number of records modified by the last function that modified the database.

Parameter

Input

Name	Type	Description
statementHandle	integer	Zero or handle to the SQL statement from `DBActivateSQL`, `DBActivateMap`, or any of the functions that return a statement handle. If `0`, returns the most recent number of records modified on any hstmt.

Return Value

Name	Type	Description
numberofRecords	integer	Number of modified records. Returns `0` if the statement is not a `SELECT` statement.

Example

```
hdbc = DBConnect("DSM=CVI32_Samples");
...
hstmt = DBActivateSQL(hdbc,"UPDATE EMP SET SALARY = SALARY * 1.1
                      WHERE DEPT='101'");
numRecs = DBNumberOfModifiedRecords(hstmt);
resCode = DBDeactivateSQL(hstmt);
resCode = DBDisconnect(hdbc);
```

See Also

`DBActivateSQL`, `DBActivateMap`

DBNumberOfRecords

```
int numberofRecords = DBNumberOfRecords (int statementHandle);
```

Purpose

Returns the number of records chosen by the `SELECT` statement. Some database systems cannot determine the number of records until you fetch the last record. If the toolkit cannot determine the number of records, this function returns `DB_CANNOT_DETERMINE_NUM_RECS`.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateSQL` or `DBActivateMap` or any of the functions that return a statement handle.

Return Value

Name	Type	Description
numberofRecords	integer	Number of records chosen by the `SELECT` statements. If less than or equal to `0`, the toolkit could not determine the number of records.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Example

```
hdbc = DBConnect("DSN=CVI32_Samples");
resCode = DBAllowFetchAnyDirection(hdbc,TRUE);
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
numRecs = DBNumberOfRecords(hstmt);
resCode = DBDeactivate(hstmt);
resCode = DBDisconnect(hdbc);
```

See Also

`DBAllowFetchAnyDirection`

DBOpenConnection

```
int status = DBOpenConnection (int connectionHandle);
```

Purpose

Opens an existing but closed connection. Unlike `DBConnect`, `DBOpenConnection`, in conjunction with `DBNewConnection`, allows you to set connection attributes before opening the connection.

Parameter

Input

Name	Type	Description
connectionHandle	integer	Connection handle that `DBNewConnection` returned. If the connection is already open, this function returns an error.

Return Value

Name	Type	Description
status	integer	Result code that `DBOpenSQLStatement` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hdbc = DBNewConnection();
resCode = DBSetConnectionAttribute(hdbc,
                                ATTR_DB_CONN_CONNECTION_TIMEOUT, 100);
resCode = DBSetConnectionAttribute(hdbc,
        ATTR_DB_CONN_ISOLATION_LEVEL, DB_ISOLATION_LEVEL_ISOLATED);
resCode = DBOpenConnection(hdbc);
...
resCode = DBCloseConnection(hdbc);
resCode = DBDiscardConnection(hdbc);
```

See Also

`DBConnect`, `DBNewConnection`, `DBSetConnectionAttribute`, `DBGetConnectionAttribute`, `DBCloseConnection`, `DBDiscardConnection`

DBOpenSchema

```
int status = DBOpenSchema (int connectionHandle, tDBSchemaType queryType,
                           VARIANT restrictions);
```

Purpose

Creates and activates a `SELECT` statement that returns schema information.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` returned.
queryType	`tDBSchemaType`	Type of schema query to run.
restrictions	`VARIANT`	Variant containing an array of variants containing restrictions on the schema query. Use `CA_DEFAULT_VAL` if you want no restrictions. Use a `NULL` variant for unrestricted elements of the array.

Return Value

Name	Type	Description
statementHandle	integer	Handle for schema query execution that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the schema query was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Example for the Restrictions Parameter

```
VARIANT restrictArray[4];
VARIANT vRestrictArray;

restrictArray[0] = CA_VariantNULL();
restrictArray[1] = CA_VariantNULL();
CA_VariantSetCString (&(restrictArray[2]), "testres");
restrictArray[3] = CA_VariantNULL();
CA_VariantSet1DArray (&vRestrictArray, CAVT_VARIANT, 4,
                      restrictArray);
```

Query Types

The following table shows the possible query types and the restrictions that might be available.

Note ***The OLE DB specification requires that providers support the*** `DP_PROVIDER_TYPES`***,*** `DB_SCHEMA_TABLES`***, and*** `DB_SCHEMA_COLUMNS` ***query types. The specification does not require that providers support any other query types or any restriction criteria.***

Table 5-3. Query Types

Query Type	Criteria values
DB_SCHEMA_ASSERTS	Constraint catalog Constraint schema Constraint name
DB_SCHEMA_CATALOGS	Catalog name
DB_SCHEMA_CHARACTER_SETS	Character set catalog Character set schema Character set name
DB_SCHEMA_COLLATIONS	Collation catalog Collation schema Collation name
DB_SCHEMA_COLUMNS	Table catalog Table schema Table name Column name

Table 5-3. Query Types (Continued)

Query Type	Criteria values
DB_SCHEMA_CHECK_CONSTRAINTS	Constraint catalog Constraint schema Constraint name
DB_SCHEMA_CONSTRAINT_COLUMN_USAGE	Table catalog Table schema Table name Column name
DB_SCHEMA_CONSTRAINT_TABLE_USAGE	Table catalog Table schema Table name
DB_SCHEMA_KEY_COLUMN_USAGE	Constraint catalog Constraint schema Constraint name Table catalog Table schema Table name Column name
DB_SCHEMA_REFERENTIAL_CONSTRAINTS	Constraint catalog Constraint schema Constraint name
DB_SCHEMA_TABLE_CONSTRAINSTS	Constraint catalog Constraint schema Constraint name Table catalog Table schema Table name Constraint type
DB_SCHEMA_COLUMN_DOMAIN_USAGE	Domain catalog Domain schema Domain name Column name
DB_SCHEMA_INDEXES	Table catalog Table schema Index name Type Table name

Table 5-3. Query Types (Continued)

Query Type	Criteria values
DB_SCHEMA_COLUMN_PRIVILEGES	Table catalog Table Schema Table name Column name Grantor Grantee
DB_SCHEMA_TABLE_PRIVILEGES	Table catalog Table schema Table name Grantor Grantee
DB_SCHEMA_USAGE_PRIVILEGES	Object catalog Object schema Object name Object type Grantor Grantee
DB_SCHEMA_PROCEDURE	Procedure catalog Procedure schema Procedure name Column name
DB_SCHEMA_SCHEMATA	Catalog name Schema name Schema owner
DB_SCHEMA_SQL_LANGUAGES	<none>
DB_SCHEMA_STATISTICS	Table catalog Table schema Table name
DB_SCHEMA_TABLES	Table catalog Table schema Table name Table type
DB_SCHEMA_TRANSLATIONS	Translation catalog Translation schema Translation name

Table 5-3. Query Types (Continued)

Query Type	Criteria values
DB_SCHEMA_PROVIDER_TYPES	Data type Best match
DB_SCHEMA_VIEWS	Table catalog Table schema Table name
DB_SCHEMA_VIEW_COLUMN_USAGE	View catalog View schema View name
DB_SCHEMA_VIEW_TABLE_USAGE	View catalog View schema View name
DB_SCHEMA_PROCEDURE_PARAMETERS	Procedure catalog Procedure schema Procedure name Parameter name
DB_SCHEMA_FOREIGN_KEYS	PK table catalog PK table schema PK table name FK table catalog FK table schema FK table name
DB_SCHEMA_PRIMARY_KEYS	PK table catalog PK table schema PK table name
DB_SCHEMA_PROCEDURE_COLUMNS	Procedure catalog Procedure schema Procedure name Column name

Example

```
VARIANT restrictArray[4];
VARIANT vRestrictArray;

/* Set up the restrictions to match only table testres. */
restrictArray[0] = CA_VariantNULL();
restrictArray[1] = CA_VariantNULL();
CA_VariantSetCString (&(restrictArray[2]), "testres");
restrictArray[3] = CA_VariantNULL();
CA_VariantSet1DArray (&vRestrictArray, CAVT_VARIANT, 4,
                      restrictArray);

/* Open the tables Schema. */
hstmt = DBOpenSchema (hdbc, DB_SCHEMA_TABLES, vRestrictArray);
resCode = DBBindColChar (hstmt, 1, 29, catalog, &catalogStat,"");
resCode = DBBindColChar (hstmt, 2, 29, schema, &schemaStat,"");
resCode = DBBindColChar (hstmt, 3, 29, name, &nameStat,"");
resCode = DBBindColInt (hstmt, 5, &guid, &guidStat);
resCode = DBBindColChar (hstmt, 6, 29, descr, &descrStat,"");

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}

hstmt = DBDeactivateSQL (hstmt);
```

DBOpenSQLStatement

```
int status = DBOpenSQLStatement (int statementHandle);
```

Purpose

Opens a SQL statement that you created with `DBNewSQLStatement`.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBNewSQLStatement` returned.

Return Value

Name	Type	Description
status	integer	Result code that `DBOpenSQLStatement` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hstmt = DBNewSQLStatement (hdbc, "SELECT UUT_NUM, MEAS1, \
                           MEAS2 FROM TESTRES");
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_CURSOR_TYPE,
                                  DB_CURSOR_TYPE_FORWARD_ONLY);
/* Set other attributes. */
...
resCode = DBOpenSQLStatement(hstmt);
...
resCode = DBCloseSQLStatement(hstmt);
```

DBPrepareSQL

```
int statementHandle = DBPrepareSQL (int connectionHandle, char
                          SQLStatement[]);
```

Purpose

Prepares a SQL statement for execution. `DBPrepareSQL` is required when the SQL statement contains parameters.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` returns.
SQLStatement	char []	Stored procedure or SQL statement to prepare.

Return Value

Name	Type	Description
statementHandle	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Input Parameter Example

```
hstmt = DBPrepareSQL (hdbc, "SELECT UUT_NUM, MEAS1, MEAS2 \
    FROM TESTRES WHERE MEAS1 > ?");
resCode = DBCreateParamDouble(hstmt, "", DB_PARAM_INPUT, 1.0);
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
/* Because there are no output parameters, you do */
/* not have to close the statement separately.   */
hstmt = DBDeactivateSQL(hstmt);
```

Output Parameter Example

```
int inParam, outParam, retParam;
...
/* This example works with Microsoft SQL Server. */

/* Remove stored procedure if it exist to        */
/* prevent an error when it is created.          */
resCode = DBImmediateSQL(hdbc, "if exists (select * from \
        sysobjects where id = object_id('dbo.sp_CVITest')\
        and sysstat & 0xf = 4) drop procedure \
        dbo.sp_AdoTest");

/* Create the stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest( \
    @InParam int, @OutParam int OUTPUT ) as select
    @OutParam = @InParam + 10 SELECT * FROM Authors \
    WHERE State <> 'CA' return @OutParam +10");

/* Force the system to execute as a stored       */
/* procedure rather than a SQL statement.       */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_CVItest");
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Create the parameters, could also use DBRefreshParams. */
resCode = DBCreateParamInt(hstmt, "", DB_PARAM_RETURN_VALUE, -1);
resCode = DBCreateParamInt(hstmt, "InParam", DB_PARAM_INPUT, 10);
resCode = DBCreateParamShort(hstmt, "OutParam", DB_PARAM_OUTPUT, -1);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by the stored procedure. */
}

/* You must close (but not deactivate) the statement */
/* before examining output parameters.               */
resCode = DBClosePreparedSQL(hstmt);

/* Get the parameters. */
resCode = DBGetParamInt(hstmt, 0, &retParam);
resCode = DBGetParamInt(hstmt, 1, &inParam);
resCode = DBGetParamInt(hstmt, 2, &outParam);
printf("return param %d input param %d output param %d\n",
       retParam, inParam, outParam);
```

```
/* Discard the statement. */
hstmt = DBDiscardSQLStatement(hstmt);
```

DBPrimaryKeys

```
int statementHandle = DBPrimaryKeys (int connectionHandle, char
                                     tableName[]);
```

Purpose

Creates and activates a `SELECT` statement that returns information about the set of columns that make up the primary keys of a table. You can then use the `DBFetch` and `DBBindCol` or `DBGetCol` functions to retrieve the information. Each record contains the following columns shown in the following table.

Note ***Not all database systems support primary keys.***

Column	Type	Description
Table Catalog	string	Table catalog.
Table Schema	string	Table schema.
Table Name	string	Table name.
Column Name	string	Column name.
Column GUID	integer	Column GUID.
Column Property ID	integer	Column property ID.
Sequence Number	short integer	Number of this column within the primary key.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returned.
TableName	char []	Pointer to a string containing the name of the table for which primary key information is selected.

Return Value

Name	Type	Description
statementHandle	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Example

```
hstmt = DBPrimaryKeys(hdbc, "testres");

resCode = DBBindColChar (hstmt, 1, 128, tableCatalog, &stat1, "");
resCode = DBBindColChar (hstmt, 2, 128, tableSchema, &stat2, "");
...
resCode = DBBindColInt (hstmt, 7, 128, seqNum, &stat7);

while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
}
resCode = DBDeactivateSQL(hstmt);
```

DBPutColBinary

```
int status = DBPutColBinary (int statementHandle, int columnNumber, void
                             *value, unsigned int sizeinBytes);
```

Purpose

Puts a binary value into the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBPutColBinary`.

Note ***You can use*** `DBPutColBinary` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record where you place the value. The first field/column number is `1`.
value	void pointer	Binary value to place in the specified field/column of the current record.
sizeinBytes	unsigned integer	Size in bytes of the binary data.

Return Value

Name	Type	Description
status	integer	Result code that `DBPutColBinary` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
unsigned char *toDBBits = NULL;
int bitsSize = 6;

toDBBits = malloc(bitsSize);
/* Writing this data as a sting would fail      */
/* because it contains embedded NUL characters. */
toDBBits[0] = 'N'; toDBBits[1] = 0; toDBBits[2] = 'C';
toDBBits[3] = 'B'; toDBBits[4] = 0; toDBBits[5] = 250;

hstmt = DBActivateSQL(hdbc, "SELECT THE BITS FROM BINTEST");

resCode = DBCreateRecord(hstmt);
resCode = DBPutColBinary(hstmt, 1, toDBBits, bitsSize);
resCode = DBPutRecord(hstmt);

resCode = DBDeactivateSQL(hstmt);
hstmt = 0;
```

See Also

DBBindColBinary, DBMapColumnToBinary

DBPutColChar

```
int status = DBPutColChar (int statementHandle, int columnNumber, char
                           value[], char *formatString);
```

Purpose

Puts a string value into the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBPutColChar`.

Note ***You can use*** `DBPutColChar` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record where you place the value. The first field/column number is `1`.
value	char []	String value to place in the specified field/column of the current record.
formatString	char []	Format string. Use the empty string, `""`, if you want the default format. See Appendix C, *Format Strings*, for details about formatting.

Return Value

Name	Type	Description
status	integer	Result code that `DBPutColChar` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, \
    LOOPNUM, MEAS1, MEAS2, CHANGER FROM REC1000");

/* Create a new record. */
resCode = DBCreateRecord(hstmt);

/* Put values into the record. */
resCode = DBPutColChar(hstmt, 1, "newrec", "");
resCode = DBPutColInt(hstmt, 2, 47);
resCode = DBPutColFloat(hstmt, 3, 23.2);
resCode = DBPutColDouble(hstmt, 4, 42.6);
resCode = DBPutColShort(hstmt, 5, 32);

/* Put the record to the database. */
resCode = DBPutRecord(hstmt);

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColChar, DBMapColumnToChar

DBPutColDouble

```
int status = DBPutColDouble (int statementHandle, int columnNumber, double
                             value);
```

Purpose

Puts a double-precision value into the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBPutColDouble`.

Note ***You can use*** `DBPutColDouble` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record where you place the value. The first field/column number is `1`.
value	double-precision	Double-precision value to place in the specified field/column of the current record.

Return Value

Name	Type	Description
status	integer	Result code that `DBPutColDouble` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, \
    LOOPNUM, MEAS1, MEAS2, CHANGER FROM REC1000");

/* Create a new record. */
resCode = DBCreateRecord(hstmt);

/* Put values into the record. */
resCode = DBPutColChar(hstmt, 1, "newrec", "");
resCode = DBPutColInt(hstmt, 2, 47);
resCode = DBPutColFloat(hstmt, 3, 23.2);
resCode = DBPutColDouble(hstmt, 4, 42.6);
resCode = DBPutColShort(hstmt, 5, 32);

/* Put the record to the database. */
resCode = DBPutRecord(hstmt);

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColDouble, DBMapColumnToDouble

DBPutColFloat

```
int status = DBPutColFloat (int statementHandle, int columnNumber, float
                            value);
```

Purpose

Puts a floating-point value into the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBPutColFloat`.

Note ***You can use*** `DBPutColFloat` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record where you place the value. The first field/column number is `1`.
value	float	Floating-point value to place in the specified field/column of the current record.

Return Value

Name	Type	Description
status	integer	Result code that `DBPutColFloat` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, \
    LOOPNUM, MEAS1, MEAS2, CHANGER FROM REC1000");

/* Create a new record. */
resCode = DBCreateRecord(hstmt);

/* Put values into the record. */
resCode = DBPutColChar(hstmt, 1, "newrec", "");
resCode = DBPutColInt(hstmt, 2, 47);
resCode = DBPutColFloat(hstmt, 3, 23.2);
resCode = DBPutColDouble(hstmt, 4, 42.6);
resCode = DBPutColShort(hstmt, 5, 32);

/* Put the record to the database. */
resCode = DBPutRecord(hstmt);

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColFloat, DBMapColumnToFloat

DBPutColInt

```
int status = DBPutColInt (int statementHandle, int columnNumber, int value);
```

Purpose

Puts an integer value into the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBPutColInt`.

☞ **Note** ***You can use*** `DBPutColInt` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record where you place the value. The first field/column number is `1`.
value	integer	Integer value to place in the specified field/column of the current record.

Return Value

Name	Type	Description
status	integer	Result code that `DBPutColInt` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, \
    LOOPNUM, MEAS1, MEAS2, CHANGER FROM REC1000");

/* Create a new record. */
resCode = DBCreateRecord(hstmt);

/* Put values into the record. */
resCode = DBPutColChar(hstmt, 1, "newrec", "");
resCode = DBPutColInt(hstmt, 2, 47);
resCode = DBPutColFloat(hstmt, 3, 23.2);
resCode = DBPutColDouble(hstmt, 4, 42.6);
resCode = DBPutColShort(hstmt, 5, 32);

/* Put the record to the database. */
resCode = DBPutRecord(hstmt);

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColInt, DBMapColumnToInt

DBPutColNull

```
int status = DBPutColNull (int statementHandle, int columnNumber);
```

Purpose

Puts a SQL Null value into the current record. Notice that SQL Null is distinct from `NULL` in C. `DBPutColNull` returns an error if the field/column specified does not allow SQL Null values.

☞ **Note** ***You can use*** `DBPutColNull` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record where you place the SQL Null. The first field/column number is `1`.

Return Value

Name	Type	Description
status	integer	Result code that `DBPutColNull` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, \
    LOOPNUM, MEAS1, MEAS2, CHANGER FROM REC1000");

/* Create a new record. */
resCode = DBCreateRecord(hstmt);

/* Put values into the record. */
resCode = DBPutColChar(hstmt, 1, "newrec", "");
resCode = DBPutColInt(hstmt, 2, 47);
/* If we do not want to save measurements, we can put SQL Null values*/
resCode = DBPutColNull(hstmt, 3);
resCode = DBPutColNull(hstmt, 4);
resCode = DBPutColShort(hstmt, 5, 32);

/* Put the record to the database. */
resCode = DBPutRecord(hstmt);

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

DBPutColShort

```
int status = DBPutColShort (int statementHandle, int columnNumber, short
                            value);
```

Purpose

Puts a short integer value into the current record. Instead of binding values and then fetching a record, you can fetch a record and then use `DBPutColShort`.

Note ***You can use*** `DBPutColShort` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record where you place the value. The first field/column number is `1`.
value	short integer	Short value to place in the specified field/column of the current record.

Return Value

Name	Type	Description
status	integer	Result code that `DBPutColShort` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Execute a select statement. */
hstmt = DBActivateSQL (hdbc, "SELECT UUT_NUM, \
    LOOPNUM, MEAS1, MEAS2, CHANGER FROM REC1000");

/* Create a new record */
resCode = DBCreateRecord(hstmt);

/* Put values into the record. */
resCode = DBPutColChar(hstmt, 1, "newrec", "");
resCode = DBPutColInt(hstmt, 2, 47);
resCode = DBPutColFloat(hstmt, 3, 23.2);
resCode = DBPutColDouble(hstmt, 4, 42.6);
resCode = DBPutColShort(hstmt, 5, 32);

/* Put the record to the database. */
resCode = DBPutRecord(hstmt);

/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColShort, DBMapColumnToShort

DBPutColVariant

```
int status = DBPutColVariant (int statementHandle, int columnNumber, VARIANT
                              value);
```

Purpose

Puts the value contained in a variant into the current record. Variants allow additional data types beyond the traditional integer, short, float, double-precision and string.

Note ***You can use*** `DBPutColVariant` ***while you use binding or mapping for other fields/columns.***

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement from `DBActivateMap`, `DBActivateSQL`, or any of the functions that return a statement handle.
columnNumber	integer	Field/column number within the record where you place the value. The first field/column number is `1`.
value	`VARIANT`	Variant containing the value to place in the specified field/column of the current record.

Return Value

Name	Type	Description
status	integer	Result code that `DBPutColVariant` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
VARIANT loopNumV;
...
/* Create a new record. */
resCode = DBCreateRecord(hstmt);
/* Put values into the record. */
...
vStatus = CA_VariantSetInt (&loopNumV, loopNum);
...
resCode = DBPutColVariant(hstmt, 2, loopNumV);
/* Put the record to the database. */
resCode = DBPutRecord(hstmt);
/* Deactivate the SQL statement. */
hstmt = DBDeactivateSQL (hstmt);
```

See Also

DBBindColDouble, DBMapColumnToDouble

DBPutRecord

```
int status = DBPutRecord (int statementHandle);
```

Purpose

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

You can override the default behavior of DBPutRecord for individual columns by setting the status variable for a column before calling DBPutRecord. If you set the status variable to DB_NULL_DATA, DBPutRecord places a SQL Null value in the column. If the status variable is DB_NO_DATA_CHANGE, DBPutRecord does not attempt to update the column. This is useful when the record contains read-only columns.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that DBActivateMap, DBActivateSQL, DBNewSQLStatement, or DBPrepareSQL returned.

Return Value

Name	Type	Description
status	integer	Result code that DBPutRecord returns. DBPutRecord returns the set of result codes listed in the function description for DBError.

Example

```
char serNum[11];
long serNumLen;
...
hstmt = DBActivateSQL(hdbc, "SELECT * FROM TESTLOG");
serialNumLen = 11;
resCode = DBBindColChar(hstmt, 1, serNum, &serNumLen);
...
resCode = DBCreateRecord(hstmt, 1);
strcpy(serNum,"PDX 600R");
...
resCode = DBPutRecord(hstmt);
         ...
resCode = DBDeactivateSQL();
```

See Also

DBActivateSQL, DBActivateMap, DBCreateRecord, DBBindCol functions, DBMapColumnTo functions

DBRefreshParams

```
int status = DBRefreshParams (int statementHandle);
```

Purpose

Retrieves provider-side parameter information for the stored procedure or parameterized query associated with the statement. Using `DBRefreshParams` can cause your program to run more slowly than when you specify the parameters with the `DBCreateParam` functions.

Not all providers or ODBC drivers support `DBRefreshParams`.

Parameter

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned.

Return Value

Name	Type	Description
status	integer	Result code that `DBRefreshParams` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
int inParam, outParam, retParam;
...
/* Create the stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest( \
        @InParam int, @OutParam int OUTPUT ) as select \
        @OutParam = @InParam + 10 SELECT * FROM Authors \
        WHERE State <> 'CA' return @OutParam +10");

/* Specify that the next command is a stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_CVItest");
if (hstmt == 0)
```

```
/* Refresh the parameters from the stored procedure.     */
/* Note: We could also create the parameters instead.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamInt(hstmt, 1, 10);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Process records returned by the stored procedure. */
}
/* You must close the statement before you can */
/* examine output parameters.                  */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameters. */
resCode = DBGetParamInt(hstmt, 0, &retParam);
resCode = DBGetParamInt(hstmt, 1, &inParam);
resCode = DBGetParamInt(hstmt, 2, &outParam);

/* Discard the statement. */
hstmt = DBDiscardSQLStatement(hstmt);
```

DBRollback

```
int status = DBRollback (int connectionHandle);
```

Purpose

Discards all changes that you make 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.

`DBRollback` discards the following changes:

- Saved changes on records other than the current record.
- Records that you created with a call to `DBCreateRecord`.
- New values that you placed in the current record with calls to `DBPutRecord`.

After rollback, you must call one of the `DBFetch` functions to position on a valid record.

Parameter

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returned.

Return Value

Name	Type	Description
status	integer	Result code that `DBRollback` returns. `DBRollback` returns the set of result codes listed in the function description for `DBError`.

Example

```
hdbc = DBConnect("DSN=QESS;UID=shawkins;SRVR=PENNY");
...
resCode = DBBeginTran(hdbc);
hstmt = DBActivateSQL(hdbc, "UPDATE EMP SET SALARY = SALARY * 1.1");
resCode = DBDeactivateSQL(hstmt);
resCode = DBRollback(hdbc);
resCode = DBDisconnect(hdbc);
```

See Also

`DBBeginTran`, `DBCommit`

DBSetAttributeDefault

```
int status = DBSetAttributeDefault (int connectionHandle, int attribute,
                                    ...);
```

Purpose

Sets the attribute value for all subsequent statements created on a connection.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returned.
attribute	integer	Attribute to set to one of the following values: `ATTR_DB_USE_COMMAND`—Sets toolkit to always use an ADO command object as well as an ADO recordset object when executing a command. `ATTR_DB_LOCK_TYPE`—Database lock type to use. `ATTR_DB_CURSOR_TYPE`—Database cursor type to use. `ATTR_DB_COMMAND_TYPE`—Command type.
value	any type	Value for the attribute; can be one of the values listed near the end of this function description.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetAttributeDefault` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Attributes for the value Parameter

Attribute	Values
`ATTR_DB_USE_COMMAND`	`DB_USE_RECORDSET_ONLY` `DB_USE_COMMAND`
`ATTR_DB_LOCK_TYPE`	Sets the lock type: `DB_LOCK_READ_ONLY`—You cannot alter the data. `DB_LOCK_PESSIMISTIC`—Provider ensures successful editing of the records, usually by locking records at the data source immediately when a user edits them. `DB_LOCK_OPTIMISTIC`—Provider locks records only when you call `DBUpdateRecord`. `DB_LOCK_BATCH_OPTIMISTIC`—Required for batch updates.

Attribute	Values
ATTR_DB_CURSOR_TYPE	Sets the cursor type. The cursor types are: DB_CURSOR_TYPE_DYNAMIC—Additions, changes, and deletions by other users are visible, and all types of movement through the recordset are allowed. DB_CURSOR_TYPE_STATIC—Static copy of a set of records Additions, changes, or deletions by other users are not visible. DB_CURSOR_TYPE_FORWARD_ONLY—Identical to a static cursor except that you can only scroll forward through records. This setting improves performance when you only have to make a single pass through a recordset. DB_CURSOR_TYPE_KEYSET—Like a dynamic cursor, except that you cannot see records that other users add. Records that other users delete are inaccessible from your recordset. Data changes by other users within records continue to be visible.
ATTR_DB_COMMAND_TYPE	Determines how the input text is interpreted: DB_COMMAND_UNKNOWN—ADO attempts to determine the command type. DB_COMMAND_TEXT—SQL statement. DB_COMMAND_TABLE—Table name. DB_COMMAND_STORED_PROC—Call to a stored procedure.

Example

```
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
```

DBSetBackwardCompatibility

```
int status = DBSetBackwardCompatibility (int version);
```

Purpose

Sets compatibility with previous versions of the LabWindows/CVI SQL Toolkit.

Version 1.*x* compatibility changes the following behavior:

- Functions that return handles, such as `DBConnect` and `DBActivateSQL`, always return zero when they fail, instead of an error code.
- `DBColumnType` coerces the column type to one of the eight data types that version 1.*x* supports, instead of the full range of types that version 2.0 supports.

Parameter

Input

Name	Type	Description
version	integer	Version number of the LabWindows/CVI SQL Toolkit. The version number is three decimal digits. The default is 200; any value below 200 sets compatibility with versions 1.0 and 1.1.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetBackwardCompatibility` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
resCode = DBSetBackwardCompatibility(110);
```

DBSetColumnAttribute

```
int status = DBSetColumnAttribute (int statementHandle, int index,
                                   tDBColumnAttr attribute, ...);
```

Purpose

Sets the value of a field/column attribute. You can set only the value attribute and in certain cases the actual size attribute. All other attributes are read only.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBNewSQLStatement` or `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL` or `DBActivateMap`.
index	integer	Index of the column. The index of the first column is `1`.
attribute	`tDBColumnAttr`	Attribute to set. Valid values appear in the Parameter Discussion.
value	any type (passed by value)	Value for the field/column attribute. The data type of the value depends on the value of the attribute.

Return Value

Name	Type	Description
status	integer	Result code that `DBGetParamAttribute` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion for attribute and value Parameters

Attribute	Type	Description
ATTR_DB_COLUMN_VALUE	VARIANT	Value of field/column.
ATTR_DB_COLUMN_ACTUAL_SIZE	long integer	Actual size reserved. Usually read only. Some Providers might allow this attribute to be set to reserve space for BLOB data.

DBSetConnectionAttribute

```
int status = DBSetConnectionAttribute (int connectionHandle,
                                       tDBConnectionAttr attribute, ...);
```

Purpose

Sets a connection attribute. Because you must set most attributes before you open the connection (except when you use `ATTR_DB_CONN_DEFAULT_DATABASE`), `DBSetConnectionAttribute` is used most often in conjunction with `DBNewConnection` and `DBOpenConnection`.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the connection that `DBNewConnection` returns or, if you can set the attribute after the connection is open, the connection that `DBConnect` returns.
attribute	`tDBConnection Attr`	Attribute to set. Some providers do not support all attributes.
value	any type (passed by value)	Value for the attribute. The type of the value varies depending on the attribute. Some providers do not support all attributes.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetConnectionAttribute` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion for attribute and value Parameters

Attribute	Type	Description
ATTR_DB_CONN_CONNECTION_STRING	string	a series of argument = value clauses separated by semicolons that describe the connection you want. The Active Data Object (ADO) standard recognizes the following arguments. All other arguments are passed directly to the provider unchanged. Provider—Name of the provider to use for the connection. Data Source—Name of the data source for the connection. User ID—User name to use when opening the connection. Password—Specifies the password to use when opening the connection. File Name—Name of a provider-specific file (for example, a persisted data source object) containing preset connection information. Remote Provider—Name of a provider to use when opening a client-side connection. Remote Server—Path name of the server to use when opening a client-side connection.
ATTR_DB_CONN_COMMAND_TIMEOUT	long integer	Number of seconds to wait for a command to execute
ATTR_DB_CONN_CONNECTION_TIMEOUT	long integer	Number of seconds to wait for a connection to be established.
ATTR_DB_CONN_DEFAULT_DATABASE	string	Name of the default database for database systems that support storing tables in multiple databases. This attribute is the only one that you can set after a connection is established.

Attribute	Type	Description
ATTR_DB_CONN_ISOLATION_LEVEL	long integer	Isolation level of the connection: DB_ISOLATION_LEVEL_CHAOS—You cannot overwrite pending changes from more highly isolated transactions. DB_ISOLATION_LEVEL_READ_UNCOMMITTED—From one transaction you can view uncommitted changes in other transactions. DB_ISOLATION_LEVEL_READ_COMMITTED—Default. From one transaction you can view changes in other transactions only after they have been committed. DB_ISOLATION_LEVEL_REPEATABLE_READ—From one transaction you cannot see changes made in other transactions, but requerying can bring new recordsets. DB_ISOLATION_LEVEL_SERIALIZABLE—Transactions take place in isolation of other transactions.
ATTR_DB_CONN_ATTRIBUTES	long integer	Attributes of the connection, the sum of one or more of the following values: DB_XACT_COMMIT_RETAINING—Performs retaining commits. In other words, calling DBCommit automatically starts a new transaction. Not all providers support this value. DB_XACT_ABORT_RETAINING—Performs retaining aborts. In other words, calling DBRollback automatically starts a new transaction. Not all providers support this value.

Attribute	Type	Description
ATTR_DB_CONN_CURSOR_LOCATION	long integer	Location of the cursor: DB_CURSOR_LOC_SERVER—Default. Uses cursors that the data provider or driver provide. DB_CURSOR_LOC_CLIENT—Uses client-side cursors supplied by a local cursor library.
ATTR_DB_CONN_MODE	long integer	Connection mode: DB_CONN_MODE_READ—Read-only permissions. DB_CONN_MODE_WRITE—Write-only permissions. DB_CONN_MODE_READ_WRITE—Read/write permissions. DB_CONN_MODE_SHARE_DENY_READ—Prevents others from opening connection with read permissions. DB_CONN_MODE_SHARE_DENY_WRITE—Prevents others from opening connection with write permissions. DB_CONN_MODE_SHARE_EXCLUSIVE—Prevents others from opening connection. DB_CONN_MODE_SHARE_DENY_NONE—Prevents others from opening connection with any permissions.
ATTR_DB_CONN_PROVIDER	string	Name of the provider of the connection. The default is MSDASQL.
ATTR_DB_CONN_STATE	long integer	Open/closed state of the connection. DB_OBJECT_STATE_CLOSED = 0 DB_OBJECT_STATE_OPEN = 1

Example

```
hdbc = DBNewConnection();
resCode = DBSetConnectionAttribute(hdbc,
                                   ATTR_DB_CONN_CONNECTION_STRING,
                                   "DSN=cvi ss;User ID=sa;Password=");
resCode = DBSetConnectionAttribute(hdbc,
                                     ATTR_DB_CONN_CONNECTION_TIMEOUT,
                                     100);
resCode = DBSetConnectionAttribute(hdbc,
                                     ATTR_DB_CONN_ISOLATION_LEVEL,
                                     DB_ISOLATION_LEVEL_SERIALIZABLE);
resCode = DBOpenConnection(hdbc);
...
resCode = DBCloseConnection(hdbc);
resCode = DBDiscardConnection(hdbc);
```

See Also

DBNewConnection, DBOpenConnection, DBGetConnectionAttribute, DBCloseConnection, DBDiscardConnection

DBSetDatabase

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

Purpose

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

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection previously that `DBConnect` or `DBNewConnection` returned.
databaseName	char []	Name of the new default database.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetDatabase` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
hdbc = DBConnect("DSN=QESS;UID=STEPHEN");
res_code = DBSetDatabase(hdbc, "TESTS")
...
res_code = DBDisconnect(hdbc);
```

DBSetParamAttribute

```
int status = DBSetParamAttribute (int statementHandle, int index,
                                  tDBParamAttr attribute, ...);
```

Purpose

Sets a parameter attribute.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle that `DBActivateSQL`. `DBActivateMap`, or `DBNewSQLStatement` returned.
index	integer	Index of the parameter. The first parameter index is `1`.
attribute	`tDBParam Attr`	Attribute to set.
value	any type (passed by value)	Value for the parameter attribute. The type of the value varies depending on the attribute.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetParamAttribute` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion for attribute and value Parameters

Attribute	Type	Description
ATTR_DB_PARAM_VALUE	VARIANT	Value of parameter.
ATTR_DB_PARAM_DIRECTION	long integer	Direction of parameter: DB_PARAM_INPUT DB_PARAM_OUTPUT DB_PARAM_INPUT_OUTPUT DB_PARAM_RETURN_VALUE DB_PARAM_UNKNOWN **Note:** Some providers cannot determine the direction of parameters to stored procedures. You cannot rely on DBRefreshParams in such cases.
ATTR_DB_PARAM_PRECISION	byte	Total number of digits.
ATTR_DB_PARAM_NUMERIC_SCALE	byte	Number of digits to the right of decimal.
ATTR_DB_PARAM_SIZE	integer	Max size in bytes.

Attribute	Type	Description
ATTR_DB_PARAM_ATTRIBUTES	long integer	Sum of zero or more of the following values: DB_PARAM_SIGNED DB_PARAM_NULLABLE DB_PARAM_LONG
ATTR_DB_PARAM_TYPE	long integer	Type of parameter. DB_EMPTY DB_TINYINT DB_SMALLINT DB_INTEGER DB_BIGINT DB_UNSIGNEDTINYINT DB_UNSIGNEDSMALLINT DB_UNSIGNEDINT DB_UNSIGNEDBIGINT DB_FLOAT DB_DOUBLEPRECISION DB_CURRENCY DB_DECIMAL DB_NUMERIC DB_BOOLEAN DB_ERROR DB_USERDEFINED DB_VARIANT DB_IDDISPATCH DB_IUNKNOWN DB_GUID DB_DATE DB_DBDATE DB_DBTIME DB_DATETIME DB_BSTR DB_CHAR DB_VARCHAR DB_LONGVARCHAR DB_WCHAR DB_VARWCHAR DB_LONGVARWCHAR DB_BINARY DB_VARBINARY DB_LONGVARBINARY

Example

```
/* This example works with Microsoft SQL server. */
/* Drop the stored procedure if it already exists. */
resCode = DBImmediateSQL(hdbc, "if exists (select * \
    from sysobjects where id = object_id('dbo.sp_AdoTest')\
    and sysstat & 0xf = 4) drop procedure dbo.sp_AdoTest");

/* Create the stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_AdoTest( \
    @InParam float, @OutParam float OUTPUT ) as select \
    @OutParam = @InParam + 1.5 SELECT * FROM Authors \
    WHERE State <> 'CA' return @OutParam +1.7");
```

```
/* Prepare a statement that calls the stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
    DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");

/* Create the parameters. */
resCode = DBCreateParamDouble(hstmt, "", DB_PARAM_RETURN_VALUE, -1);
resCode = DBCreateParamDouble(hstmt, "", DB_PARAM_INPUT_OUTPUT, 10);
resCode = DBCreateParamDouble(hstmt, "OutParam", DB_PARAM_OUTPUT,
                              -1);

/* Modify the second parameter. */
resCode = DBSetParamAttribute (hstmt, 2, ATTR_DB_PARAM_DIRECTION,
                               DB_PARAM_INPUT);
resCode = DBSetParamAttribute (hstmt, 2, ATTR_DB_PARAM_VALUE,
                               CA_VariantDouble (42.42));

/* Execute the prepared statement. */
resCode = DBExecutePreparedSQL(hstmt);
/* Close the statement.  The statement must be closed  */
/* before you can examine the output parameters.       */
resCode = DBClosePreparedSQL(hstmt);
```

DBSetParamBinary

```
int status = DBSetParamBinary (int statementHandle, int index, void *value,
                               int sizeinBytes);
```

Purpose

Sets the value of a parameter for a stored procedure or SQL statement that you prepared with `DBPrepareSQL`. Typically, you need `DBSetParamBinary` only when you use `DBRefreshParams`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle from `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement`.
index	integer	Index of the parameter.
value	void pointer	Binary value for the parameter.
sizeinBytes	integer	Size of the binary variable in bytes.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetParamBinary` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
unsigned char data[10];

data[0]='I';data[1]=0;data[2]='h';data[3]='a';data[4]='t';
data[5]='e';data[6]=0;data[7]='A';data[8]='D';data[9]='O';
hstmt = DBPrepareSQL (hdbc, "SELECT BUMMER, MEAS1,\
        MEAS2 FROM TESTRES WHERE BUMMER = ?");
resCode = DBRefreshParams(hstmt);
resCode = DBSetParamBinary(hstmt, 1, data, 10);
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBGetParamBinary, DBGetParamBinaryBuffer, DBClosePreparedSQL

DBSetParamChar

```
int status = DBSetParamChar (int statementHandle, int index, char *value,
                             char *formatString);
```

Purpose

Sets the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Typically, you need `DBSetParamChar` only when you use `DBRefreshParams`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement`.
index	integer	Index of the parameter.
value	char *	String value for the parameter.
formatString	char []	Format string. Use the empty string, `""`, if you want the default format. See Appendix C, *Format Strings*, for details about formatting.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetParamChar` returns. Use `DBErrorMessage` to obtain the text of the error message. This function returns the set of result codes listed in the `DBError` function description.

Example

```
hstmt = DBPrepareSQL (hdbc, "SELECT UUT_NUM, MEAS1,\
        MEAS2 FROM TESTRES WHERE UUT_NUM = ?");
resCode = DBRefreshParams(hstmt);
resCode = DBSetParamChar(hstmt, 1, "yyd2860b1", "");
resCode = DBExecutePreparedSQL(hstmt);
```

```
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    ...
}
hstmt = DBDeactivateSQL(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBGetParamChar, DBGetParamCharBuffer, DBClosePreparedSQL

DBSetParamDouble

```
int status = DBSetParamDouble (int statementHandle, int index, double value);
```

Purpose

Sets the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Typically, you need `DBSetParamDouble` only when you use `DBRefreshParams`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement`.
index	integer	Index of the parameter.
value	double-precision	Double-precision value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetParamDouble` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam float, @OutParam float OUTPUT ) as  \
        select @OutParam = @InParam + 10.1 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam \
        +10.1");
```

```
/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");

/* Set command type back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamDouble(hstmt, 1, 10.5);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* process records returned by the stored procedure. */
}

/* Close the statement.  Output parameters are invalid.  */
/* until you close the statement.                        */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamDouble(hstmt, 0, &retParam);
resCode = DBGetParamDouble(hstmt, 1, &inParam);
resCode = DBGetParamDouble(hstmt, 2, &outParam);

/* Discard the statement. */
hstmt = DBDiscardSQLStatement(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBGetParamDouble, DBClosePreparedSQL

DBSetParamFloat

```
int status = DBSetParamFloat (int statementHandle, int index, float value);
```

Purpose

Sets the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Typically, you need `DBSetParamFloat` only when you use `DBRefreshParams`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement`.
index	integer	Index of the parameter.
value	float	Floating-point value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetParamFloat` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam float, @OutParam float OUTPUT ) as  \
        select @OutParam = @InParam + 10.1 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam \
        +10.1");

/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
```

```
/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");

/* Set command type back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamFloat(hstmt, 1, 10.5);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* process records returned by the stored procedure */
}

/* Close the statement.  Output parameters are invalid.  */
/* until you close the statement.                        */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamFloat(hstmt, 0, &retParam);
resCode = DBGetParamFloat(hstmt, 1, &inParam);
resCode = DBGetParamFloat(hstmt, 2, &outParam);

/* Discard the statement. */
hstmt = DBDiscardSQLStatement(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBGetParamFloat, DBClosePreparedSQL

DBSetParamInt

```
int status = DBSetParamInt (int statementHandle, int index, int value);
```

Purpose

Sets the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Typically, you need `DBSetParamInt` only when you use `DBRefreshParams`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement`.
index	integer	Index of the parameter.
value	integer	Integer value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetParamInt` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam int, @OutParam int OUTPUT ) as  \
        select @OutParam = @InParam + 10 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam +10");

/* Prepare a statement that calls the stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
```

```
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamInt(hstmt, 1, 10);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* process records returned by the stored procedure */
}

/* Close the statement.  Output parameters are invalid.  */
/* until you close the statement.                        */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamInt(hstmt, 0, &retParam);
resCode = DBGetParamInt(hstmt, 1, &inParam);
resCode = DBGetParamInt(hstmt, 2, &outParam);

/* Discard the statement. */
hstmt = DBDiscardSQLStatement(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBGetParamInt, DBClosePreparedSQL

DBSetParamShort

```
int status = DBSetParamShort (int statementHandle, int index, short value);
```

Purpose

Sets the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Typically, you need `DBSetParamShort` only when you use `DBRefreshParams`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returned. You cannot use a statement handle from `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement`.
index	integer	Index of the parameter.
value	short integer	Short integer value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetParamShort` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam int, @OutParam int OUTPUT ) as  \
        select @OutParam = @InParam + 10 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam +10");

/* Prepare a statement that calls the stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");
```

```
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
resCode = DBSetParamShort(hstmt, 1, 10);

    /* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
    while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
            /* Process records returned by the stored procedure. */
    }

/* Close the statement.  Output parameters are invalid.  */
/* until you close the statement.                        */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamShort(hstmt, 0, &retParam);
resCode = DBGetParamShort(hstmt, 1, &inParam);
resCode = DBGetParamShort(hstmt, 2, &outParam);

/* Discard the statement. */
hstmt = DBDiscardSQLStatement(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBGetParamShort, DBClosePreparedSQL

DBSetParamVariant

```
int status = DBSetParamVariant (int statementHandle, int index, float value);
```

Purpose

Sets the value of a parameter for a stored procedure or SQL statement that you prepare with `DBPrepareSQL`. Variants allow for more types than the traditional integer, short, float, double-precision and string.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBPrepareSQL` returns. You cannot use a statement handle from `DBActivateSQL`, `DBActivateMap`, or `DBNewSQLStatement`.
index	integer	Index of the parameter.
value	float	Variant containing the value for the parameter.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetParamVariant` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Create a stored procedure. */
resCode = DBImmediateSQL(hdbc, "create proc sp_CVITest(
        @InParam float, @OutParam float OUTPUT ) as  \
        select @OutParam = @InParam + 10.1 SELECT * FROM \
        Authors WHERE State <> 'CA' return @OutParam \
        +10.1");
```

```
/* Set the command type attribute to stored procedure. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_STORED_PROC);

/* Prepare a statement that calls the stored procedure. */
hstmt = DBPrepareSQL (hdbc, "sp_Adotest");

/* Set command type back to the default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_COMMAND_TYPE,
                                DB_COMMAND_UNKNOWN);

/* Refresh the parameters from the stored procedure.    */
resCode = DBRefreshParams(hstmt);

/* Set the input parameter. */
CA_VariantSetInt (&inParamV, 10);
resCode = DBSetParamVariant(hstmt, 2, inParamV);

/* Execute the statement. */
resCode = DBExecutePreparedSQL(hstmt);
...
/* Close the statement.  Output parameters are invalid.  */
/* until you close the statement.                        */
resCode = DBClosePreparedSQL(hstmt);

/* Examine the parameter values. */
resCode = DBGetParamVariant(hstmt, 0, &retParamV);
resCode = DBGetParamVariant(hstmt, 1, &inParamV);
resCode = DBGetParamVariant(hstmt, 2, &outParamV);

/* Discard the statement. */
hstmt = DBDiscardSQLStatement(hstmt);
```

See Also

DBPrepareSQL, DBRefreshParams, DBExecutePreparedSQL, DBGetParamFloat, DBClosePreparedSQL

DBSetStatementAttribute

```
int status = DBSetStatementAttribute (int statementHandle, tDBStatementAttr
                                      attribute, ...);
```

Purpose

Sets a SQL statement attribute. You can set attributes for statements created with `DBActivateSQL`, `DBActivateMap`, `DBNewSQLStatement` and `DBPrepareSQL`. For attributes that you must set before you open the SQL statement, you must set the statement attribute as follows:

1. Call `DBNewSQLStatement` or `DBPrepareSQL`.
2. Set the attribute.
3. Call `DBOpenSQLStatement` or `DBExecuteSQL`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBActivateSQL`, `DBActivateMap`, `DBNewSQLStatement`, or `DBPrepareSQL` returns.
attribute	`tDBStatementAttr`	Attribute to set. Some providers do not support all attributes.
value	any type (passed by value)	Value for the parameter attribute. The type of the value varies depending on the attribute. Some providers do not support all attributes.

Return Value

Name	Type	Description
status	integer	Result code that `DBSetStatementAttribute` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Parameter Discussion for attribute and value Parameters

Attribute	Type	Description
`ATTR_DB_STMT_PAGE_SIZE`	long integer	Sets the number of records in a page.
`ATTR_DB_STMT_ABSOLUTE_PAGE`	long integer	Move to the specified page. Page numbers start at one.
`ATTR_DB_STMT_ABSOLUTE_POSITION`	long integer	Moves the to the specified record number. Record numbers start at one.
`ATTR_DB_STMT_CACHE_SIZE`	long integer	Sets the number of records the provider keeps in its in memory buffer and the number of records the provider retrieves at one time.

Attribute	Type	Description
ATTR_DB_STMT_CURSOR_TYPE	tDBCursorType	Sets the cursor type. The following list shows cursor types: DB_CURSOR_TYPE_DYNAMIC—Additions, changes, and deletions by other users are visible, and all types of movement through the recordset are allowed. DB_CURSOR_TYPE_STATIC—Static copy of a set of records Additions, changes, or deletions by other users are not visible. DB_CURSOR_TYPE_FORWARD_ONLY—Identical to a static cursor except that you can only scroll forward through records. This setting improves performance when you only have to make a single pass through a recordset. DB_CURSOR_TYPE_KEYSET—Like a dynamic cursor, except that you cannot see records that other users add. Records that other users delete are inaccessible from your recordset. Data changes by other users within records continue to be visible.

Attribute	Type	Description
ATTR_DB_STMT_CURSOR_LOCATION	tDBCursorLoc	Sets the cursor location: DB_CURSOR_LOC_SERVER—Uses cursors that the data provider or driver provide. These cursors are sometimes very flexible and allow for some additional sensitivity to reflecting changes that others make to the actual data source. DB_CURSOR_LOC_CLIENT—Uses client-side cursors supplied by a local cursor library. Local cursor engines often allow many features that driver-supplied cursors might not.
ATTR_DB_STMT_LOCK_TYPE	tDBLockType	Sets the lock type: DB_LOCK_READ_ONLY—You cannot alter the data. DB_LOCK_PESSIMISTIC—Provider ensures successful editing of the records, usually by locking records at the data source immediately when a user edits them. DB_LOCK_OPTIMISTIC—Provider locks records only when you call DBUpdateRecord. DB_LOCK_BATCH_OPTIMISTIC—Required for batch updates.
ATTR_DB_STMT_MAX_RECORDS	long integer	Sets the maximum number of records the provider returns from the data source. If 0, the provider returns all records. This parameter is read-only after you open or execute a statement.

Attribute	Type	Description
ATTR_DB_STMT_MARSHAL_OPTIONS	tDBMarshalOpt	Specifies how to write modified data back to the server: DB_MARSHAL_OPT_ALL—All records are written back to the server. DB_MARSHAL_OPT_MODIF_ONLY—Only modified data is written back to the server.
ATTR_DB_STMT_BOOKMARK	Variant	Moves to the record that the bookmark indicates.
ATTR_DB_STMT_COMMAND_TYPE	tDBCommandType	Specifies how to interpret the input text: DB_COMMAND_UNKNOWN—ADO attempts to determine the command type. DB_COMMAND_TEXT—SQL statement. DB_COMMAND_TABLE—Table name. DB_COMMAND_STORED_PROC—Cal l to a stored procedure.
ATTR_DB_STMT_COMMAND_TIMEOUT	long integer	Time in seconds to wait for a command to execute. If 0, wait for an unlimited time period.
ATTR_DB_STMT_PREPARED	long integer	Whether the command to save a prepared (compiled) version of the statement to speed future executions of the statement. Applies to statements created with DBPrepareSQL.
ATTR_DB_STMT_ACTIVE_CONNECTION	String	String defining a connection. The same as the connection string used for DBConnect. Applies only to statements created with DBPrepareSQL.

Attribute	Type	Description
ATTR_DB_STMT_NAME	String	Name of the command. Applies only to statements created with DBPrepareSQL.
ATTR_DB_STMT_FILTER	Variant	Used to filter screen out records in a recordset. The variant can contain: - A criteria string made up of individual clauses connected by AND or OR. - An array of bookmarks. - A filter group value. DB_FILTER_NONE—Removes the current filter. DB_FILTER_PENDING—Only records that have changed but have not yet been sent to the server. Only applicable for batch update mode. DB_FILTER_AFFECTED—Only records affected by the last DBDeleteRecord or DBUpdateBatch. DB_FILTER_FETCHED—Records in the current cache.

Example

```
hstmt = DBNewSQLStatement (hdbc, "SELECT UUT_NUM, MEAS1, \
                           MEAS2 FROM TESTRES");
/* Set the cursor type. */
resCode = DBSetStatementAttribute(hstmt,
    ATTR_DB_STMT_CURSOR_TYPE, DB_CURSOR_TYPE_FORWARD_ONLY);
...
resCode = DBOpenSQLStatement(hstmt);
/* Set the absolute position. */
resCode = DBSetStatementAttribute(hstmt,
    ATTR_DB_STMT_ABSOLUTE_POSITION, 2);
...
resCode = DBCloseSQLStatement(hstmt);
```

Bookmark and Filter Example

```
VARIANT vFilterArray;
int filterIndex = 0;
VARIANT bookmarks[2];

hstmt = DBNewSQLStatement (hdbc, "SELECT UUT_NUM, MEAS1, \
                                  MEAS2 FROM TESTRES");
resCode = DBOpenSQLStatement(hstmt);

/* Criteria string example.                              */
/* Note: It almost always much more efficient to use a   */
/* where clause in the SQL statement, instead of using a */
/* criteria string in a filter.                          */
CA_VariantSetCString (&filter, "(MEAS1 > 1.0)");
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_FILTER, filter);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Only records where MEAS1 greater than 1.0. */
    ...
}

/* Filter constant example. */
CA_VariantSetLong (&filter, DB_FILTER_NONE);
resCode = DBSetStatementAttribute(hstmt, ATTR_DB_STMT_FILTER,
                                  filter);
i = 0;
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Will get all records. */
    ...
    /* Get some bookmarks to use in next example. */
    if ((i == 0) || (i == 2)) {
      resCode = DBGetStatementAttribute(hstmt, ATTR_DB_STMT_BOOKMARK,
                                        &(bookmarks[filterIndex++]));
    i++;
    ...
}

/* Bookmark array example. */
CA_VariantSet1DArray (&vFilterArray, CAVT_VARIANT, 2, bookmarks);
resCode = DBSetStatementAttribute(hstmt,
    ATTR_DB_STMT_FILTER, vFilterArray);
while ((resCode = DBFetchNext (hstmt)) == DB_SUCCESS) {
    /* Will only get records at the bookmarks we   */
    /* obtained in the previous example.           */
    ...
}
resCode = DBCloseSQLStatement(hstmt);
```

DBSources

```
int status = DBSources (int sourceType);
```

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. `DBSources` does not support the `SELECT` information functions. You do not have to connect to a a database before using `DBSources`. Each record contains the following columns:

Column	Type	Description
Name	string	Data source name.
Description	string	Data source description.

Parameter

Input

Name	Type	Description
sourceType	integer	`DB_AVAILABLE` is the only supported option.

Return Value

Name	Type	Description
statementHandle	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Example

```
hstmt = DBSources(DB_SRC_AVAILABLE);
resCode = DBBindColChar(hstmt, 1, 32, srcName, &srcNameStat, "");
resCode = DBBindColChar(hstmt, 2, 256, remarks, &remarksStat,"");
...
/* Fetch records. */
...
resCode = DBDeactivateSQL(hstmt);
```

See Also

DBActivateSQL, DBActivateMap

DBTables

```
int status = DBTables (int connectionHandle, char tableCatalog[], char
                           tableSchema[], char tableName[], 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. Each record contains the following columns:

Column	Type	Description
Table Catalog	string	Table catalog.
Table Schema	string	Table schema.
Table Name	string	Table name.
Table Type	string	Table type.
Table GUID	integer	Table GUID.
Description	string	Table description.

Parameters

Input

Name	Type	Description
connectionHandle	integer	Handle to the database connection that `DBConnect` or `DBNewConnection` returns.
tableCatalog	char []	Table catalog. Use this parameter to restrict the table information that this function returns to a single catalog. Some ADO providers and ODBC drivers do not support this parameter.
tableSchema	char []	Table schema. Use this parameter to restrict the table information that this function returns to a single schema. Some ADO providers and ODBC drivers do not support this parameter.

Name	Type	Description
tableName	char []	Table name. Use this parameter to restrict the table information that this function returns to a single table. Some ADO providers and ODBC drivers do not support this parameter.
flags	integer	Specifies the type(s) of table(s) for which information will return.

Return Value

Name	Type	Description
statementHandle	integer	Statement execution handle that identifies the statement and is a parameter to other functions. If less than or equal to `0`, the toolkit was not able to execute the statement.

Note ***Prior to version 2.0, the LabWindows/CVI SQL Toolkit always returned*** `0` ***on error. To minimize changes to programs that depend on this behavior, set the compatibility mode to version 1.1 with the following function call:***
`DBSetBackwardCompatibility(110);`

Parameter Discussion

You can use any of the following values:

Constant	Value	Description
`DB_TBL_TABLE`	0x0001	Table names.
`DB_TBL_VIEW`	0x0002	View names.
`DB_TBL_PROCEDURE`	0x0004	Stored procedure 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.

Example

```
hstmt = DBTables(hdbc,"","","", DB_TBL_TABLE | DB_TBL_VIEW);
resCode = DBBindColChar(hstmt, 1, 128, tabCat, &tabCatStat);
resCode = DBBindColChar(hstmt, 2, 128, tabSchema, &schemaStat, "");
resCode = DBBindColChar(hstmt, 3, 128, table, &tableStat, "");
resCode = DBVindColChar(hstmt, 4, 128, tabType, &tabStat, "");
resCode = DBBindColInt(hstmt, 5, &tabGUID, &tabGUIDStat);
resCode = DBBindColChar(hstmt, 6, remarks, &remStat, "");
...
/* Fetch records. */
...
resCode = DBDeactivateSQL(hstmt);
```

See Also

DBActivateSQL, DBActivateMap

DBUpdateBatch

```
int status = DBUpdateBatch (int statementHandle, tDBAffect
                            affectwhichRecords);
```

Purpose

Writes pending changes to the database. You can only use `DBUpdateBatch` with statements that you have opened with lock type of `DB_LOCK_BATCH_OPTIMISTIC`.

Parameters

Input

Name	Type	Description
statementHandle	integer	Handle to the SQL statement that `DBActivateSQL` or `DBActivateMap` returns.
affectwhichRecords	`tDBAffect`	Indicates records to update. Choices are `DB_AFFECT_CURRENT`, the current record only, or `DB_AFFECT_ALL`, all records for which changes are pending.

Return Value

Name	Type	Description
status	integer	Result code that `DBUpdateBatch` returns. This function returns the set of result codes listed in the function description for `DBError`. Use `DBErrorMessage` to obtain the text of the error message.

Example

```
/* Set lock type attribute to batch optimistic. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_LOCK_TYPE,
                                DB_LOCK_BATCH_OPTIMISTIC);
hstmt = DBActivateSQL(hdbc, "SELECT * FROM rec1000");
/* Bind variables. */
...
```

```
for (i = 0; i < 1000; i++) {
    /* Create the new record. */
    resCode = DBCreateRecord (hstmt);
/* Set bound variables. */
    ...
    /* Insert the record into the database. */
    resCode = DBPutRecord (hstmt);
}
resCode = DBUpdateBatch(hstmt, DB_AFFECT_ALL);
resCode = DBDeactivateSQL(hstmt);
/* Set lock type attribute back to default. */
resCode = DBSetAttributeDefault(hdbc, ATTR_DB_LOCK_TYPE,
                                DB_LOCK_OPTIMISTIC);
```

DBWarning

```
int status = DBWarning (void);
```

Purpose

Returns the warning generated by the last SQL Toolkit Library function you called. You normally call `DBWarning` after calling `DBError`, to determine whether the database system or the last function returned warnings.

Return Value

Name	Type	Description
status	integer	Result code, either a warning code that the database system returns or a constant value. Refer to the *Return Codes* section.

Return Codes

Constant	Value	Description
`DB_NULL_DATA`	`-2`	Toolkit function returned a null value.
`DB_TRUNCATION`	`-1`	Toolkit function truncated the returned value because the size of the value exceeded the buffer.

See Also

`DBError`

A

SQL Language Reference

This appendix briefly explains SQL commands, operators, and functions. This version of SQL is included in the ODBC standard and 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)

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)

Table A-2. SQL Objects (Continued)

Object	Description	Examples
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*

Table A-3. SQL Clauses (Continued)

Name/Syntax	Applicable Commands	Description	Examples
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"
.T. .F.	Date-time constant.	{2/8/60},{16:59:59}
	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')

Table A-4. SQL Operators (Continued)

Operator Class and Operators	Description	Examples
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		WHERE *uut_num* = ANY (SELECT...)
BETWEEN	Compare with list of rows.	WHERE *meas1* BETWEEN 0.0 AND 1.0
EXISTS	Within value range.	WHERE EXISTS (SELECT...)
[NOT] LIKE		WHERE *uut_num* LIKE 'TEK%'
[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*) (*must be numeric column*)	Average of all non-null values in *column_name*.
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*.

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.

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`. The last section in this appendix explains some error messages that might be vague or misleading.

The following table lists error codes that the LabWindows/CVI SQL Toolkit can return.

Table B-1. SQL Toolkit Error Code

Error Code	Error Constant Name	Description/Cause
-299	DB_CS_OUT_OF_MEMORY	Thread error, out of memory.
-298	DB_CS_INVALID_PARAMETER	Thread error, invalid parameter.
-297	DB_CS_SYSTEM_ERROR	Thread error, system error.
-296	DB_CS_NOT_INIT_OR_DELETED	Thread error, critical section either not initialized or deleted.
-295	DB_CS_NOT_OWNED	Thread error, critical section not owned.
-294	DB_CS_ALREADY_INIT	Thread error, critical section already initialized.
-96	DB_BETA_EXPIRED	Beta version of LabWindows/CVI SQL Toolkit expired.
-94	DB_CANT_FIND_STMT	Cannot find statement. Invalid statement or connection handle.
-93	DB_CANT_FIND_CONNECTION	Cannot find connection. Invalid connection handle.
-92	DB_CANT_DETERMINE_NUM_RECS	The ADO provider or ODBC driver cannot determine the number of records.
-91	DB_SYNONYM_NOT_SUPPORTED	`DB_TBL_SYNONYM` flag is not supported for `DBTables`.

Table B-1. SQL Toolkit Error Code (Continued)

Error Code	Error Constant Name	Description/Cause
-90	DB_UNKNOWN_TABLES_FLAG	Unknown flag value for DBTables.
-89	DB_CANNOT_FIND_MAP	Invalid map handle.
-88	DB_UNEXPECTED_NULL_PTR	Illegal pointer to NULL in attribute function.
-87	DB_INVALID_ATTRIBUTE	Specified attribute number is not a valid attribute number.
-87	DB_INVALID_COLUMN	Invalid field/column number.
-85	DB_UNSUPPORTED_TYPE	Conversion to unsupported type requested in DBGetVariantArrayColumns.
-83	DB_INVALID_PARAMETER_ ATTRIBUTE	Specified parameter attribute number is not a valid parameter attribute number.
-81	DB_INVALID_COLUMN_ATTRIBUTE	Specified attribute number is not a valid column attribute number.
-80	DB_COLUMN_ATTRIBUTE_READ_ONLY	Specified column attribute is a read only attribute.
-79	DB_STMT_ATTR_READ_ONLY	Specified statement attribute is a read only attribute.
-78	DB_INVALID_STMT_ATTRIBUTE	Specified attribute number is not valid for statements.
-77	DB_INVALID_FOR_COMMAND	Specified statement attribute is invalid for statements based on command objects, in other words, statements you create with DBPrepareSQL.
-76	DB_INVALID_FOR_RECORDSET	Specified statement attribute is invalid for statements based on recordset objects, in other words, statements you create with DBActivateSQL, DBActivateMap, or DBNewSQLStatement.

Table B-1. SQL Toolkit Error Code (Continued)

Error Code	Error Constant Name	Description/Cause
-75	DB_REQUIRES_RECORDSET	Specified statement attribute requires an open recordset object. You must call DBExecutePreparedSQL or DBOpenSQLStatement before you can set the specified attribute.
-74	DB_REQUIRES_COMMAND	Specified statement attribute requires and open command object. Either the statement does not use a command object or you must call DBExecutePreparedSQL before you can set the specified attribute.
-73	DB_CONN_ATTR_READ_ONLY	Specified connection attribute is read only.
-72	DB_INVALID_CONN_ATTRIBUTE	Specified connection attribute number is not a valid connection attribute number.
-71	DB_FORMAT_ERROR	Syntax error in format string.
-70	DB_FORMAT_IGNORE_INCOMPAT	Specified format string is not compatible with the data type.
-69	DB_FORMAT_IGNORE_NO_DATA_FORMAT	Specified format string contains no data formatting.
-68	DB_START_REC_TOO_BIG	Start record number is larger than the number of records.
-67	DB_FIELD_NUMBER_TOO_BIG	Field/column number is larger than the number of fields available.
-66	DB_NOT_ENOUGH_RECORDS	Start record number plus number of records requested is greater than number of available records.
-64	DB_UNSUPPORTED_SOURCE_TYPE	DB_SRC_AVAILABLE is the only value allowed for Source Type.
-63	DB_NEGATIVE_SIZE	Negative size not permitted.

Table B-1. SQL Toolkit Error Code (Continued)

Error Code	Error Constant Name	Description/Cause
-12	DB_ODBC_ERROR	ODBC error detected in DBSources. Call DBNativeError to get the underlying error code.
-11	DB_AUTOMATION_ERROR	Error detected by CVI Automation. Call DBNativeError to get the underlying error code.
-10	DB_DBSYS_ERROR	Error detected by ADO. Call DBNativeError to get the underlying error code.
-9	DB_NO_DATA_CHANGED	Value could not be changed because the field/column is read only. You can use this value as a status variable value to prevent attempts to change data in read only columns.
-5	DB_EOF	End of file.
-4	DB_USER_CANCELLED	User cancelled the operation.
-3	DB_OUT_OF_MEMORY	Out of memory.
-2	DB_NULL_DATA	Value contains SQL Null data. For DBGetVariantArrayColumn, at least one value in the field was SQL Null, the remaining columns have been ignored.
-1	DB_TRUNCATION	Value retrieved from the database was truncated.
0	DB_SUCCESS	Operation complete successfully.

The following table lists error codes for CVI Automation.

Table B-2. Error Codes for CVI Automation

Error Code	Error Constant Name	Description/Cause
-2147746306	E_CVIAUTO_INVALID_TYPE_DESC	Invalid type description.
-2147746307	E_CVIAUTO_INVALID_RETURN_TYPE	Invalid return type.
-2147746308	E_CVIAUTO_USE_CAVT_TYPE_DESC	CVI Automation CAVT style type description rather than OLE style VT type description required.
-2147746309	E_CVIAUTO_INVALID_NUM_DIM	Invalid number of dimensions.
-2147746310	E_CVIAUTO_DIFF_SAFEARRAY_TYPE	Differing safe array types.
-2147746312	E_CVIAUTO_VARIANT_NOT_SAFEARRAY	Variant is not a safe array.
-2147746313	E_CVIAUTO_NULL_RET_VAL_PARAM	NULL return parameter.
-2147746314	E_CVIAUTO_DLL_LOAD_FAILED	DLL load failed.
-2147746315	E_CVIAUTO_BAD_DLL_VERSION	DLL version is invalid.
-2147746315	E_CVIAUTO_COULD_NOT_CREATE_ MUTEX	Thread error, could not create mutex.

The following table lists error codes for OLE (Object Linking and Embedding).

Table B-3. Error Codes for OLE

ErrorCode	Description/Cause
-214748647	Not implemented.
-214783646	Ran out of memory.
-214783645	One or more arguments are invalid.
-214783644	No such interface supported.
-214783643	Invalid pointer.
-214783642	Invalid handle.

Table B-3. Error Codes for OLE (Continued)

ErrorCode	Description/Cause
-214783641	Operation aborted.
-214783640	Unspecified error.
-214783639	General access denied error.
-214783638	The data necessary to complete this operation is not yet available.
-214783663	Not implemented.
-214783662	No such interface supported.
-214783661	Invalid pointer.
-214783660	Operation aborted.
-214783659	Unspecified Error
-214783658	Thread local storage failure.
-214783657	Get shared memory allocator failure.
-214783656	Get memory allocator failure.
-214783655	Unable to initialize class cache.
-214783654	Unable to initialize RPC services.
-214783653	Cannot set thread local storage channel control.
-214783652	Could not allocate thread local storage channel control.
-214783651	User supplied memory allocator is unacceptable.
-214783650	OLE service mutex already exists.
-214783649	OLE service file mapping already exists
-214783648	Unable to map view of file for OLE service.
-214783647	Failure attempting to launch OLE service.
-214783646	There was an attempt to call `CoInitialize` a second time while single threaded.
-214783645	Remote activation was necessary but was not allowed.
-214783644	Remote activation was necessary by the server name provided was invalid.

Table B-3. Error Codes for OLE (Continued)

ErrorCode	Description/Cause
-214783643	Class is configured to run as a security ID different from the caller.
-214783642	The use of Ole1 services requiring DDE windows is disabled.
-214783641	A RunAs specification must be *domain name\user name* or simply *user name*.
-214783640	Server process could not be started. The pathname may be incorrect.
-214783639	Server process could not be started as the configured identity. The pathname may be incorrect or unavailable.
-214783638	Server process could not be started because the configured identity is incorrect. Check the username and password.
-214783637	Client is not allowed to launch this server.
-214783636	Service providing this server could not be started.
-214783635	Computer was unable to communicate with the computer providing the server.
-214783634	Server did not respond after being launched.
-214783633	Registration information for this server is inconsistent of incomplete.
-214783632	Registration information for this interface is inconsistent or incomplete.
-214783631	Operation attempted is not supported.
-2147418113	Catastrophic failure.
-2147024891	General access denied error.
-2147024890	Invalid handle.
-2147024882	Ran out of memory.
-2147024809	One or more arguments are invalid.
-2147217920	Invalid accessor.
-2147217919	Creating another row would have exceeded the total number of active rows supported by the rowset.
-2147217918	Unable to write with a read-only accessor.
-2147217917	Given values violate the database schema.

Table B-3. Error Codes for OLE (Continued)

ErrorCode	Description/Cause
-2147217916	Invalid row handle.
-2147217915	An object was open.
-2147217914	Invalid chapter.
-2147217913	A literal value in the command could not be converted to the correct type due to a reason other than data overflow.
-2147217912	Invalid binding info.
-2147217911	Permission denied.
-2147217910	Specified column does not contain bookmarks or chapters.
-2147217909	Some cost limits were rejected.
-2147217908	No command has been set for the command object.
-2147217907	Unable to find a query plan within the given cost limit.
-2147217906	Invalid bookmark.
-2147217905	Invalid lock mode.
-2147217904	No value given for one or more required parameters.
-2147217903	Invalid column ID.
-2147217902	Invalid ration.
-2147217901	Invalid value.
-2147217900	Command contained one or more errors.
-2147217899	Executing command cannot be cancelled.
-2147217898	Provider does not support the specified dialect.
-2147217897	A data source with the specified name already exists.
-2147217896	Rowset was built over a live data feed and cannot be restarted.
-2147217895	No key matching the described characteristics could be found within the current range.
-2147217894	Ownership of this tree has been given to the provider.
-2147217893	Provider is unable to determine identity for newly inserted rows.

Table B-3. Error Codes for OLE (Continued)

ErrorCode	Description/Cause
-2147217892	No nonzero weights specified for any goals supported, so goal was rejected; current goal was not changed.
-2147217891	Requested conversion is not supported.
-2147217890	lRowOffset would position you past either end of the rowset, regardless of the cRows value specified; cRowsObtained is 0.
-2147217889	Information was requested for a query, and the query was not set.
-2147217888	Provider called a method from IRowsetNotify in the consumer and NT.
-2147217887	Errors occurred.
-2147217886	A non-NULL controlling IUnknown was specified and the object being created does not support aggregation.
-2147217885	A given HROW referred to a hard- or soft-deleted row.
-2147217884	Rowset does not support fetching backwards.
-2147217883	All HROWs must be released before new ones can be obtained.
-2147217882	One of the specified storage flags was not supported.
-2147217880	Specified status flag was not `DBCOLUMNSTATUS_OK` or `DBCOLUMNSTATUS_ISNULL`.
-2147217879	Rowset cannot scroll backwards.
-2147217878	Invalid region handle.
-2147217877	Specified set of rows was not contiguous to or overlapping the rows in the specified watch region.
-2147217876	Transition from ALL* to MOVE* or EXTEND* was specified.
-2147217875	Specified region is not a proper subregion of the region identified by the given watch region handle.
-2147217874	Provider does not support multi-statement commands.
-2147217873	Specified value violated the integrity constraints for a column or table.
-2147217872	Given type name was unrecognized.
-2147217871	Execution aborted because a resource limit has been reached; no results have been returned.

Table B-3. Error Codes for OLE (Continued)

ErrorCode	Description/Cause
-2147217870	Cannot clone a command object whose command tree contains a rowset or rowsets
-2147217869	Cannot represent the current tree as text.
-2147217868	Specified index already exists.
-2147217867	Specified index does not exist.
-2147217866	Specified index was in use.
-2147217865	Specified table does not exist.
-2147217864	The rowset was using optimistic concurrency and the value of a column has been changed since it was last read.
-2147217863	Error were detected during the copy.
-2147217862	A specified precision was invalid.
-2147217861	A specified scale was invalid.
-2147217860	Invalid table ID.
-2147217859	Specified type was invalid.
-2147217858	Column ID occurred more than once in the specification.
-2147217857	Specified table already exists.
-2147217856	Specified table was in use.
-2147217855	Specified locale ID was not supported.
-2147217854	Specified record number is invalid.
-2147217853	Although the bookmark was validly formed, no row could be found to match it.
-2147217852	Value of the property was invalid.
-2147217851	Rowset was not chaptered.
-2147217850	Invalid accessor.
-2147217849	Invalid storage flags.
-2147217848	By-ref accessors are not supported by this provider.
-2147217847	Null accessors are not supported by this provider.

Table B-3. Error Codes for OLE (Continued)

ErrorCode	Description/Cause
-2147217846	Command was not prepared.
-2147217845	Specified accessor was not a parameter accessor.
-2147217844	Given accessor was write only.
-2147217843	Authentication failed.
-2147217842	Change was cancelled during notification; no columns are changed.
-2147217841	Rowset was single-chaptered and the chapter was not released.
-2147217840	Invalid source handle.
-2147217839	Provider cannot drive parameter info and `SetParameterInfo` has not been called.
-2147217838	Data source object is already initialized.
-2147217837	Provider does not support this method.
-2147217836	Number of rows with pending changes has exceeded the set limit.
-2147217835	Specified column did not exist.
-2147217834	Changes are pending on a row with a reference count of zero.
-2147217833	A literal value in the command overflowed the range of the type of the associated column.
-2147217832	Supplied HRESULT was invalid.
-2147217831	Supplied LookupID was invalid.
-2147217830	Supplied DynamicErrorID was invalid.
-2147217829	Unable to get visible data for a newly-inserted row that has not yet been updated.
-2147217828	Invalid conversion on flag.
-2147217827	Given parameter name was unrecognized.
-2147217826	Multiple storage objects can not be open simultaneously.
265920	Fetching requested number of rows would have exceeded total number of active rows supported.

Table B-3. Error Codes for OLE (Continued)

ErrorCode	Description/Cause
265921	One or more column types are incompatible; conversion errors will occur during copying.
265922	Parameter type information has been overridden by caller.
265923	Skipped bookmark for deleted or non-member row.
265924	Errors found in validating tree.
265925	There are no more rowsets.
265926	Reached start or end of rowset or chapter.
265927	Provider re-executed the command.
265928	Variable data buffer full.
265929	There are no more results.
265930	Server cannot release or downgrade a lock until the end of the transaction.
265931	Specified weight was not supported or exceeded the supported limit and was set of 0 or the supported limit.
265933	Input dialect was ignored and text was returned in different dialect.
265934	Consumer is uninterested in receiving further notification calls for this phase.
265935	Consumer is uninterested in receiving further notification calls for this phase.
265937	In order to reposition to the start of the rowset, the provider had to re-execute the query; either the order of the columns changed or columns were added to or removed from the rowset.
265938	Method had some errors; errors have been returned in the error array.
265939	Invalid row handle.
265940	A given HROW referred to a hard-deleted row.
265941	Provider was unable to keep track or all the changes; the client must refetch the data associated with the watch region using another method.
265942	Execution stopped because a resource limit has been reached; results obtained so far have been returned but execution cannot be resumed.

Table B-3. Error Codes for OLE (Continued)

ErrorCode	Description/Cause
265944	A lock was upgraded from the value specified.
265945	One or more properties were changed as allowed by provider.
265946	Errors occurred.
265947	A specified parameter was invalid.
265948	Updating this row caused more than one row to be updated in the data source.

Error Messages That Might Be Vague or Misleading

If you misspell a field/column name in a call to a `DBMapColumnTo` function, some database systems, including Microsoft Access, interpret the misspelled name as parameter. When you then call a `DBFetch` function Access returns the following error:

```
Too few parameters. Expected 1.
```

If you execute a `SELECT` statement with `DBImmediateSQL`, you cannot access the selected records, and some database systems do not release the locks for tables in the `SELECT` statement.

The vague error code `Errors Occurred` might appear, for example, when you bind, map, or put an integer or floating point value into a string field/column that is too small to contain the value.

It is difficult to discover the cause of `Undefined Error`. This error can occur if you request the value of a connection attribute from a statement instead of a connection. This error can also occur if you attempt to write records to a Microsoft SQL Server table that contains only `double` or `float` columns. Because of the problems with exact comparison of floating-point values, SQL Server does not consider floating-point columns when it constructs its internal `WHERE` clause to to uniquely identify a record. Thus, in a table that contains only floating point columns, SQL Server has no way to uniquely identify records.

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 –156.20348	$210.60 ($156.20)
`GN`	153 1.875	153 1.875
`0[S/1000]`	12567 199	12 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

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

Symbol	Description	Example Output
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
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
/ - . : , <space>	Output the character.	
\<character>	Output the character following the ‘\’ character.	\U\T\C is UTC
"<string>" '<string>'	Output the string.	“UTC” is UTC
GD	General format for dates (the “Short Date Format” in the international section of the Windows control panel). **Note:** Do not combine other format symbols with GD except [US].	

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

Symbol	Description	Example Output
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"). **Note:** 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). **Note:** 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". **Note:** 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. **Note:** 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>"` `'<string>'`	Outputs the 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]`.

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

Symbol	Description
`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]`.
`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.

D

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

This section describes the types of electronic support that National Instruments offers: bulletin board, FTP, fax-on-demand, and e-mail.

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Québec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
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	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the 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.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___) _____Phone (___) _____

Computer brand_____ Model _____Processor_____

Operating system (include version number) _____

Clock speed _____MHz RAM _____MB Display adapter _____

Mouse ___yes ___no Other adapters installed_____

Hard disk capacity _____MB Brand_____

Instruments used _____

_____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is: _____

_____

_____

_____

_____

List any error messages: _____

_____

_____

The following steps reproduce the problem: _____

_____

_____

_____

_____

LabWindows/CVI SQL Toolkit Reference Manual Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use 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.

National Instruments Products

Hardware revision _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

National Instruments software _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Other Products

Computer make and model _____

Microprocessor _____

Clock frequency or speed _____

Type of video board installed _____

Operating system version _____

Operating system mode _____

Programming language _____

Programming language version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

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 for Windows 95/98/NT*

Edition Date: September 1998

Part Number: 320960B-01

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

_____

_____

_____

_____

_____

_____

_____

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

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

E-Mail Address _____

Phone (___) _____ Fax (___) _____

Mail to: Technical Publications
National Instruments Corporation
11500 North Mopac Expressway
Austin, Texas 78759 USA

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Glossary

Prefix	Meanings	Value
k-	kilo-	10^3
M-	mega-	10^6

ADO — Active Data Object.

automatic SQL — Method for activating simple SQL `SELECT` and `CREATE TABLE` statements in the LabWindows/CVI SQL Toolkit.

BLOB — Binary Large Object, for use with binary data functions.

explicit SQL — Method for activating complex SQL `SELECT` statements and other types of statements in the LabWindows/CVI SQL Toolkit.

ODBC — Open Database Connectivity. A standard for accessing various database systems, such as Paradox, Excel, and Access.

SQL — Structured Query Language; pronounced “sequel.” A language that provides a standard interface for conducting transactions with relational database management systems, such as Paradox, Excel, and Access.

Index

A

activating SQL statements, 2-5, 3-3 to 3-5
ADO (Active Data Object)
    standard, 2-4
    support, 1-2
advanced connection functions. *See* connection attribute functions.
alternative data access functions
    DBGetColBinary, 5-86 to 5-87
    DBGetColBinaryBuffer, 5-88 to 5-89
    DBGetColChar, 5-90 to 5-91
    DBGetColCharBuffer, 5-92 to 5-93
    DBGetColDouble, 5-94 to 5-95
    DBGetColFloat, 5-96 to 5-97
    DBGetColInt, 5-98 to 5-99
    DBGetColShort, 5-100 to 5-101
    DBGetColVariant, 5-106 to 5-107
    DBPutColBinary, 5-204 to 5-205
    DBPutColChar, 5-206 to 5-207
    DBPutColDouble, 5-208 to 5-209
    DBPutColFloat, 5-210 to 5-211
    DBPutColInt, 5-212 to 5-213
    DBPutColNull, 5-214 to 5-215
    DBPutColShort, 5-216 to 5-217
    DBPutColVariant, 5-218 to 5-219
    examples, 4-4 to 4-6
    functions for getting data as an array,
        DBFreeVariantArray, 5-84 to 5-85
        DBGetVariantArray, 4-4, 5-149 to 5-150
        DBGetVariantArrayColumn, 4-4 to 4-5, 5-151 to 5-154
        DBGetVariantArrayValue, 4-4 to 4-5, 5-155 to 5-157
attribute functions. *See* statement attribute functions.
automatic SQL (maps) functions
    activating SQL statements, 2-5
    DBActivateMap, 3-4, 5-5 to 5-6
    DBBeginMap, 3-3, 5-10 to 5-11
    DBCreateTableFromMap, 3-4, 5-61 to 5-62
    DBDeactivateMap, 5-65
    DBMapColumnToBinary, 5-164 to 5-165
    DBMapColumnToChar, 3-3, 5-167 to 5-169
    DBMapColumnToDouble, 3-3, 5-170 to 5-172
    DBMapColumnToFloat, 5-173 to 5-175
    DBMapColumnToInt, 5-176 to 5-178
    DBMapColumnToShort, 5-179 to 5-181
    generating SQL statements, 3-3 to 3-5

B

bulletin board support, D-1 to D-2

C

character operators, SQL (table), A-4
clauses, SQL (table), A-3 to A-4
commands, SQL, 2-2, A-1 to A-2
comparison operators, SQL (table), A-5
compatibility with version 1.1 of the toolkit, 4-6
    DBGetSQLToolkitVersion, 5-139
    DBSetBackwardCompatibility, 5-228
compound statements, 4-2
connecting to databases, 2-5, 3-3
connection attribute functions
    DBCloseConnection, 5-30
    DBDiscardConnection, 5-68
    DBGetConnectionAttribute, 5-108 to 5-113
    DBNewConnection, 4-1, 5-185
    DBOpenConnection, 4-1, 5-191
    DBSetAttributeDefault, 5-225 to 5-227

DBSetConnectionAttribute, 4-1, 5-231 to 5-235
examples, 4-1 to 4-2
connection functions
DBConnect, 3-3, 5-40 to 5-41
DBDisconnect, 3-3, 5-70
DBSetDatabase, 5-236
constant operators, SQL (table), A-4
CREATE TABLE command, 2-2, A-1
customer communication, *xiii*, D-1
CVI automation error codes (table), B-5

D

data access functions, alternative. *See* alternative data access functions.
Data Sources Dialog Box, 2-3
data types supported by SQL Toolkit (table), 2-2
database drivers for ODBC, 1-2
database sessions, 2-5 to 2-6
activating SQL statements, 2-5
connecting to databases, 2-5, 3-3
deactivating SQL statements, 2-6
disconnecting from databases, 2-6
processing SQL statements, 2-5 to 2-6
steps in database sessions, 2-5 to 2-6
databases
concepts, 2-1 to 2-2
data types supported by SQL Toolkit (table), 2-2
tables, 2-1 to 2-2
date operators, SQL (table), A-5
date/time format strings (table), C-2 to C-4
DBActivateMap function
activating maps, 3-4
description, 5-5 to 5-6
DBActivateSQL function
description, 5-7 to 5-8
example, 3-5
DBAllowFetchAnyDirection function, 5-9
DBBeginMap function
description, 5-10 to 5-11
example, 3-3
DBBeginTran function
description, 5-10 to 5-11
example, 3-11
DBBindColBinary function, 5-14 to 5-15
DBBindColChar function
description, 5-16 to 5-18
example, 3-5
DBBindColDouble function
description, 5-19 to 5-20
example, 3-5 to 3-6
DBBindColFloat function, 5-21 to 5-22
DBBindColInt function, 5-23 to 5-24
DBBindColShort function, 5-25 to 5-26
DBCancelRecordChanges function, 5-27 to 5-28
DBClearParam function, 5-29
DBCloseConnection function, 5-30
DBClosePreparedSQL function
description, 5-31 to 5-32
example, 4-3
DBCloseSQLStatement function
description, 5-33 to 5-34
example, 4-2
DBColumnName function
description, 5-35
example, 3-10
DBColumnType function
description, 5-36 to 5-37
example, 3-10
DBColumnWidth function, 5-38
DBCommit function
description, 5-39
example, 3-11
DBConnect function
description, 5-40 to 5-41
example, 3-3
DBCreateParamBinary function, 5-42 to 5-44
DBCreateParamChar function, 5-45 to 5-47

DBCreateParamDouble function, 5-48 to 5-50
DBCreateParamFloat function, 5-51 to 5-52
DBCreateParamInt function, 5-53 to 5-55
DBCreateParamShort function, 5-56 to 5-58
DBCreateRecord function
    description, 5-59 to 5-60
    example, 3-7
DBCreateTableFromMap function
    description, 5-61 to 5-62
    example, 3-4
DBDatabases function
    description, 5-63 to 5-64
    purpose, 3-9
DBDeactivateMap function, 5-65
DBDeactivateSQL function
    description, 5-66
    freeing system resources, 3-5
DBDeleteRecord function
    description, 5-67
    example, 3-9
DBDiscardConnection function, 5-68
DBDiscardSQLStatement function
    description, 5-69
    example, 4-2
DBDisconnect function
    description, 5-70
    example, 3-3
DBError function
    description, 5-71
    example, 3-12
DBErrorMessage function
    description, 5-72
    example, 3-12
DBExecutePreparedSQL function
    description, 5-73 to 5-74
    example, 4-2
DBFetchNext function
    description, 5-75
    example, 3-6
DBFetchPrev function
    description, 5-76 to 5-77
    example, 3-6
DBFetchRandom function
    description, 5-78 to 5-79
    example, 3-6
DBForeignKeys function, 5-80 to 5-82
DBFree function, 5-83
DBGetColBinary function, 5-86 to 5-87
DBGetColBinaryBuffer function, 5-88 to 5-89
DBGetColChar function, 5-90 to 5-91
DBGetColCharBuffer function, 5-92 to 5-93
DBGetColDouble function, 5-94 to 5-95
DBGetColFloat function, 5-96 to 5-97
DBGetColInt function, 5-98 to 5-99
DBGetColShort function, 5-100 to 5-101
DBGetColumnAttribute function, 5-102 to 5-105
DBGetColVariant function, 5-106 to 5-107
DBGetConnectionAttribute function, 5-108 to 5-113
DBGetParamAttribute function, 5-114 to 5-117
DBGetParamBinary function, 5-118 to 5-120
DBGetParamBinaryBuffer function, 5-121 to 5-123
DBGetParamChar function, 5-124 to 5-125
DBGetParamCharBuffer function, 5-126 to 5-128
DBGetParamDouble function, 5-129 to 5-130
DBGetParamFloat function, 5-131 to 5-132
DBGetParamInt function, 5-133 to 5-134
DBGetParamShort function, 5-135 to 5-136
DBGetParamVariant function, 5-137 to 5-138
DBGetSQLToolkitVersion function, 5-139
DBGetStatementAttribute function, 5-140 to 5-148
DBGetVariantArray function
    description, 5-149 to 5-150
    example, 4-4

DBGetVariantArrayColumn function
description, 5-151 to 5-154
example, 4-4 to 4-5
DBGetVariantArrayValue function
description, 5-155 to 5-157
example, 4-4 to 4-5
DBImmediateSQL function
description, 5-158 to 5-159
example, 3-6
DBIndexes function, 5-160 to 5-162
DBInit function, 5-163
DBMapColumnToBinary function, 5-164 to 5-165
DBMapColumnToChar function
description, 5-167 to 5-169
example, 3-3
DBMapColumnToDouble function
description, 5-170 to 5-172
example, 3-3
DBMapColumnToFloat function, 5-173 to 5-175
DBMapColumnToInt function, 5-176 to 5-178
DBMapColumnToShort function, 5-179 to 5-181
DBMoreResults function, 5-182 to 5-183
DBNativeError function, 5-184
DBNewConnection function
description, 5-185
example, 4-1
DBNewSQLStatement function
description, 5-186 to 5-187
example, 4-2
DBNumberOfColumns function
description, 5-188
example, 3-10
DBNumberOfModifiedRecords function
description, 5-189
example, 3-10
DBNumberOfRecords function
description, 5-190
example, 3-10
DBOpenConnection function
description, 5-191
example, 5-191
DBOpenSchema function, 5-192 to 5-197
DBOpenSQLStatement function
description, 5-198
example, 4-2
DBPrepareSQL function
description, 5-199 to 5-201
example, 4-2
DBPrimaryKeys function, 5-202 to 5-203
DBPutColBinary function, 5-204 to 5-205
DBPutColChar function, 5-206 to 5-207
DBPutColDouble function, 5-208 to 5-209
DBPutColFloat function, 5-210 to 5-211
DBPutColInt function, 5-212 to 5-213
DBPutColNull function, 5-214 to 5-215
DBPutColShort function, 5-216 to 5-217
DBPutColVariant function, 5-218 to 5-219
DBPutRecord function
description, 5-220 to 5-221
inserting records, 3-7
updating records, 3-8
DBRefreshParams function, 5-222 to 5-223
DBRollback function
description, 5-224
example, 3-11
DBSetAttributeDefault function, 5-225 to 5-227
DBSetBackwardCompatibility function, 5-228
DBSetColumnAttribute function, 5-229 to 5-230
DBSetConnectionAttribute function
description, 5-231 to 5-235
example, 4-1
DBSetDatabase function, 5-236

DBSetParamAttribute function, 5-237 to 5-240
DBSetParamBinary function, 5-241 to 5-242
DBSetParamChar function, 5-243 to 5-244
DBSetParamDouble function, 5-245 to 5-246
DBSetParamFloat function, 5-247 to 5-248
DBSetParamInt function, 5-249 to 5-250
DBSetParamShort function, 5-251 to 5-252
DBSetParamVariant function, 5-253 to 5-254
DBSetStatementAttribute function
    description, 5-255 to 5-261
    example, 4-2
DBSources function
    description, 5-262 to 5-263
    purpose, 3-9
DBTables function
    description, 5-264 to 5-266
    purpose, 3-9
DBUpdateBatch function, 5-267 to 5-268
DBWarning function, 5-269
deactivating SQL statements, 2-6
DELETE command, 2-2, A-1
deleting. *See also* insert/delete/update records functions.
    records, 3-8 to 3-9
    tables, 3-9
disconnecting from databases, 2-6, 3-3
documentation
    conventions used in manual, *xii-xiii*
    organization of manual, *xi-xii*
    related documentation, *xiii*
DROP TABLE command, A-1

E

electronic support services, D-1 to D-2
e-mail support, D-2
error checking, 3-12
error codes, 4-6, B-1 to B-13
    CVI automation codes (table), B-5
    OLE codes, B-5 to B-13
    SQL Toolkit error codes (table), B-1 to B-4
error functions
    DBError, 3-12, 5-71
    DBErrorMessage, 3-12, 5-72
    DBNativeError, 5-184
    DBWarning, 5-269
explicit SQL functions
    activating SQL statements, 2-5, 3-5 to 3-6
    DBActivateSQL, 3-5, 5-7 to 5-8
    DBBindColBinary, 5-14 to 5-15
    DBBindColChar, 3-5, 5-16 to 5-18
    DBBindColDouble, 3-5 to 3-6, 5-19 to 5-20
    DBBindColFloat, 5-21 to 5-22
    DBBindColInt, 5-23 to 5-24
    DBBindColShort, 5-25 to 5-26
    DBDeactivateSQL, 3-5, 5-66
    DBImmediateSQL, 3-6, 5-158 to 5-159

F

fax and telephone support numbers, D-2
Fax-on-Demand support, D-2
fetch records functions
    DBAllowFetchAnyDirection, 5-9
    DBFetchNext, 3-6, 5-75
    DBFetchPrev, 3-6, 5-76 to 5-77
    DBFetchRandom, 3-6, 5-78 to 5-79
fetching records, 3-6 to 3-7
FOR UPDATE OF clause, SQL (table), A-4
format strings
    date/time (table), C-2 to C-4
    examples (table), C-1
    numeric (table), C-5 to C-6
FROM clause, SQL (table), A-3
FTP support, D-1
functions, SQL (table), A-6 to A-7

functions for SQL Toolkit
alternative data access functions
DBGetColBinary, 5-86 to 5-87
DBGetColBinaryBuffer, 5-88 to 5-89
DBGetColChar, 5-90 to 5-91
DBGetColCharBuffer, 5-92 to 5-93
DBGetColDouble, 5-94 to 5-95
DBGetColFloat, 5-96 to 5-97
DBGetColInt, 5-98 to 5-99
DBGetColShort, 5-100 to 5-101
DBGetColVariant, 5-106 to 5-107
DBPutColBinary, 5-204 to 5-205
DBPutColChar, 5-206 to 5-207
DBPutColDouble, 5-208 to 5-209
DBPutColFloat, 5-210 to 5-211
DBPutColInt, 5-212 to 5-213
DBPutColNull, 5-214 to 5-215
DBPutColShort, 5-216 to 5-217
DBPutColVariant, 5-218 to 5-219
functions for getting data as an array,
DBFreeVariantArray, 5-84 to 5-85
DBGetVariantArray, 4-4, 5-149 to 5-150
DBGetVariantArrayColumn, 4-4 to 4-5, 5-151 to 5-154
DBGetVariantArrayValue, 4-4 to 4-5, 5-155 to 5-157
automatic SQL (maps) functions
DBActivateMap, 3-6, 5-5 to 5-6
DBBeginMap, 3-3, 5-10 to 5-11
DBCreateTableFromMap, 3-4, 5-61 to 5-62
DBDeactivateMap, 5-65
DBMapColumnToBinary, 5-164 to 5-165
DBMapColumnToChar, 3-3, 5-167 to 5-169
DBMapColumnToDouble, 3-3, 5-170 to 5-172
DBMapColumnToFloat, 5-173 to 5-175
DBMapColumnToInt, 5-176 to 5-178
DBMapColumnToShort, 5-179 to 5-181
compatibility functions
DBGetSQLToolkitVersion, 5-139
DBSetBackwardCompatibility, 5-228
connection attribute functions
DBCloseConnection, 5-30
DBDiscardConnection, 5-68
DBGetConnectionAttribute, 5-108 to 5-113
DBNewConnection, 4-1, 5-185
DBOpenConnection, 4-1, 5-191
DBSetAttributeDefault, 5-225 to 5-227
DBSetConnectionAttribute, 4-1, 5-231 to 5-235
connection functions
DBConnect, 3-3, 5-40 to 5-41
DBDisconnect, 3-3, 5-70
DBSetDatabase, 5-236
error functions
DBError, 3-12, 5-71
DBErrorMessage, 3-12, 5-72
DBNativeError, 5-184
DBWarning, 5-269
explicit SQL functions
DBActivateSQL, 3-5, 5-7 to 5-8
DBBindColBinary, 5-14 to 5-15
DBBindColChar, 3-5, 5-16 to 5-18
DBBindColDouble, 3-5 to 3-6, 5-19 to 5-20
DBBindColFloat, 5-21 to 5-22
DBBindColInt, 5-23 to 5-24
DBBindColShort, 5-25 to 5-26

DBDeactivateSQL, 3-5, 5-66
DBImmediateSQL, 3-6, 5-158 to 5-159
fetch records functions
DBAllowFetchAnyDirection, 5-9
DBFetchNext, 3-6, 5-75
DBFetchPrev, 3-6, 5-76 to 5-77
DBFetchRandom, 3-6, 5-78 to 5-79
function summary (figure), 3-2
function tree (table), 5-1 to 5-4
information functions
DBColumnName, 3-10, 5-35
DBColumnType, 3-10, 5-36 to 5-37
DBColumnWidth, 5-38
DBDatabases, 3-9, 5-63 to 5-64
DBForeignKeys, 5-80 to 5-82
DBGetColumnAttribute, 5-102 to 5-105
DBIndexes, 5-160 to 5-162
DBNumberOfColumns, 3-10, 5-188
DBNumberOfModifiedRecords, 3-10, 5-189
DBNumberOfRecords, 3-10, 5-190
DBOpenSchema, 5-192 to 5-197
DBPrimaryKeys, 5-202 to 5-203
DBSetColumnAttribute, 5-229 to 5-230
DBSources, 3-9, 5-262 to 5-263
DBTables, 3-9, 5-264 to 5-266
initialize threading function
DBInit, 5-163
insert/delete/update records functions
DBCancelRecordChanges, 5-27 to 5-28
DBCreateRecord, 3-7, 5-59 to 5-60
DBDeleteRecord, 3-9, 5-67
DBPutRecord, 3-7, 3-8, 5-220 to 5-221
DBUpdateBatch, 5-267 to 5-268
memory function
DBFree, 5-83
obsolete functions
DBAllowFetchAnyDirection function, 5-9
DBClearParam function, 5-29
parameterized SQL statement functions
DBClosePreparedSQL, 4-3, 5-31 to 5-32
DBCreateParamBinary, 5-42 to 5-44
DBCreateParamChar, 5-45 to 5-47
DBCreateParamDouble, 5-48 to 5-50
DBCreateParamFloat, 5-51 to 5-52
DBCreateParamInt, 5-53 to 5-55
DBCreateParamShort, 5-56 to 5-58
DBExecutePreparedSQL, 4-2, 5-73 to 5-74
DBGetParamAttribute, 5-114 to 5-117
DBGetParamBinary, 5-118 to 5-120
DBGetParamBinaryBuffer, 5-121 to 5-123
DBGetParamChar, 5-124 to 5-125
DBGetParamCharBuffer, 5-126 to 5-128
DBGetParamDouble, 5-129 to 5-130
DBGetParamFloat, 5-131 to 5-132
DBGetParamInt, 5-133 to 5-134
DBGetParamShort, 5-135 to 5-136
DBGetParamVariant, 5-137 to 5-138
DBPrepareSQL, 4-2, 5-199 to 5-201
DBRefreshParams, 5-222 to 5-223
DBSetParamAttribute, 5-237 to 5-240
DBSetParamBinary, 5-241 to 5-242
DBSetParamChar, 5-243 to 5-244
DBSetParamDouble, 5-245 to 5-246
DBSetParamFloat, 5-247 to 5-248
DBSetParamInt, 5-249 to 5-250
DBSetParamShort, 5-251 to 5-252
DBSetParamVariant, 5-253 to 5-254

SQL statement functions
DBCloseSQLStatement, 4-2, 5-33 to 5-34
DBDiscardSQLStatement, 4-2, 5-69
DBGetStatementAttribute, 5-140 to 5-148
DBMoreResults, 5-182 to 5-183
DBNewSQLStatement, 4-2, 5-186 to 5-187
DBOpenSQLStatement, 4-2, 5-198
DBSetStatementAttribute, 4-2, 5-255 to 5-261
transaction functions
DBBeginTran, 3-11, 5-12 to 5-13
DBCommit, 3-11, 5-39
DBRollback, 3-11, 5-224

G

GROUP BY clause, SQL (table), A-3

H

HAVING clause, SQL (table), A-4

I

information functions
data source information, 3-9 to 3-10
DBColumnName, 3-10, 5-35
DBColumnType, 3-10, 5-36 to 5-37
DBColumnWidth, 5-38
DBDatabases, 3-9, 5-63 to 5-64
DBForeignKeys, 5-80 to 5-82
DBGetColumnAttribute, 5-102 to 5-105
DBIndexes, 5-160 to 5-162
DBNumberOfColumns, 3-10, 5-188
DBNumberOfModifiedRecords, 3-10, 5-189
DBNumberOfRecords, 3-10, 5-190
DBOpenSchema, 5-192 to 5-197
DBPrimaryKeys, 5-202 to 5-203
DBSetColumnAttribute, 5-229 to 5-230
DBSources, 3-9, 5-262 to 5-263
DBTables, 3-9, 5-264 to 5-266
SELECT statement information, 3-10 to 3-11
initialize threading function (DBInit), 5-163
INSERT command, 2-2, A-1
insert/delete/update records functions
DBCancelRecordChanges, 5-27 to 5-28
DBCreateRecord, 3-7, 5-59 to 5-60
DBDeleteRecord, 3-9, 5-67
DBPutRecord, 3-7, 3-8, 5-220 to 5-221
DBUpdateBatch, 5-267 to 5-268
inserting records, 3-7 to 3-8
installation of LabWindows/CVI SQL Toolkit, 1-1 to 1-2
ODBC database driver files, 1-2
procedure, 1-1 to 1-2

L

LabWindows/CVI SQL Toolkit. *See* SQL Toolkit.
logical operators, SQL (table), A-5

M

manual. *See* documentation.
memory deallocation (DBFree function), 5-83
multithreading, 4-6
DBInit, 5-163

N

numeric format strings (table), C-5 to C-6
numeric operators, SQL (table), A-4

O

Object, Active Data (ADO)
standard, 2-4
support, 1-2
Object Linking and Embedding (OLE) error codes, B-5 to B-13
objects, SQL (table), A-2 to A-3
obsolete functions
DBAllowFetchAnyDirection function, 5-9
DBClearParam, 5-29
ODBC Administrator, 2-3 to 2-4
ODBC Database Drivers
driver file installation, 1-2
setting up with ODBC Administrator, 2-3 to 2-4
third party drivers, 2-4
ODBC (Open Database Connectivity) Standard, 2-3
OLE error codes, B-5 to B-13
operators, SQL (table), A-4 to A-6
ORDER BY clause, SQL (table), A-4

P

parameterized SQL statement functions
DBClosePreparedSQL, 4-3, 5-31 to 5-32
DBCreateParamBinary, 5-42 to 5-44
DBCreateParamChar, 5-45 to 5-47
DBCreateParamDouble, 5-48 to 5-50
DBCreateParamFloat, 5-51 to 5-52
DBCreateParamInt, 5-53 to 5-55
DBCreateParamShort, 5-56 to 5-58
DBExecutePreparedSQL, 4-2, 5-73 to 5-74
DBGetParamAttribute, 5-114 to 5-117
DBGetParamBinary, 5-118 to 5-120
DBGetParamBinaryBuffer, 5-121 to 5-123
DBGetParamChar, 5-124 to 5-125
DBGetParamCharBuffer, 5-126 to 5-128
DBGetParamDouble, 5-129 to 5-130
DBGetParamFloat, 5-131 to 5-132
DBGetParamInt, 5-133 to 5-134
DBGetParamShort, 5-135 to 5-136
DBGetParamVariant, 5-137 to 5-138
DBPrepareSQL, 4-2, 5-199 to 5-201
DBRefreshParams, 5-222 to 5-223
DBSetParamAttribute, 5-237 to 5-240
DBSetParamBinary, 5-241 to 5-242
DBSetParamChar, 5-243 to 5-244
DBSetParamDouble, 5-245 to 5-246
DBSetParamFloat, 5-247 to 5-248
DBSetParamInt, 5-249 to 5-250
DBSetParamShort, 5-251 to 5-252
DBSetParamVariant, 5-253 to 5-254
examples, 4-2 to 4-3
portability issues, 1-3
processing SQL statements, 2-5 to 2-6

R

records. *See also* insert/delete/update records functions.
deleting, 3-8 to 3-9
fetching, 3-6 to 3-7
inserting, 3-7 to 3-8
updating, 3-8

S

SELECT command, 2-2, A-2
SELECT statement
functions for returning information, 3-10 to 3-11
processing, 2-5 to 2-6
set operators, SQL (table), A-5

SQL statements. *See also* specific statements.
activating, 2-5
automatic SQL, 2-5, 3-3 to 3-5
explicit SQL, 2-5, 3-5 to 3-6
deactivating, 2-6
processing, 2-5 to 2-6
SQL (Structured Query Language)
clauses (table), A-3 to A-4
commands, 2-2, A-1 to A-2
definition, 2-2
functions (table), A-6 to A-7
objects (table), A-2 to A-3
operators (table), A-4 to A-5
SQL Toolkit. *See also* functions for SQL Toolkit.
automatic SQL (maps), 3-3 to 3-5
connecting to databases, 3-3
deleting
records, 3-8 to 3-9
tables, 3-9
error checking, 3-12
error codes (table), B-1 to B-4
explicit SQL statements, 3-5 to 3-6
features, 1-3
fetching records, 3-6 to 3-7
information functions
data source information, 3-9 to 3-10
SELECT statement information, 3-10 to 3-11
inserting records, 3-7 to 3-8
installation, 1-1 to 1-2
overview, 1-3
portability issues, 1-3
transactions, 3-11 to 3-12
updating records, 3-8
SQL statement functions
DBCloseSQLStatement, 4-2, 5-33 to 5-34
DBDiscardSQLStatement, 4-2, 5-69
DBGetStatementAttribute, 5-140 to 5-148
DBMoreResults, 5-182 to 5-183
DBNewSQLStatement, 4-2, 5-186 to 5-187
DBOpenSQLStatement, 4-2, 5-198
DBSetStatementAttribute, 4-2, 5-255 to 5-261
examples, 4-2
statements, parameterized SQL. *See* parameterized SQL statement functions.
statements, SQL. *See* SQL statements.
Structured Query Language (SQL). *See* SQL (Structured Query Language).

T

tables
data types supported by SQL Toolkit (table), 2-2
deleting, 3-9
purpose and use, 2-1 to 2-2
technical support, D-1 to D-2
telephone and fax support numbers, D-2
third party ODBC Database Drivers, 2-4
time format strings (table), C-2 to C-4
transaction functions
DBBeginTran, 3-11, 5-12 to 5-13
DBCommit, 3-11, 5-39
DBRollback, 3-11, 5-224
transactions, 3-10 to 3-11
tree of SQL functions (table), 5-1 to 5-4

U

UPDATE command, 2-2, A-2
updating records, 3-8. *See also* insert/delete/update records functions.

W

WHERE clause, SQL (table), A-3