

# **Introduction to Programming DB-Library for Visual Basic Help**

To use this Help file, choose the Search button, and then type the word you want to search for. You can also use the previous topic (<<) and next topic (>>) buttons to browse through the topics in this Help file.

# Before You Begin

The programming components in Microsoft® SQL Server™ 6.0 enable you to develop sophisticated client/server applications that tap the power of Microsoft SQL Server from workstations running the Microsoft® Windows NT™, Microsoft® Windows®, and MS-DOS® operating systems.

Microsoft® DB-Library™ is used to create client applications for SQL Server.

# About *Programming DB-Library for Visual Basic*

This book describes Microsoft DB-Library for the Microsoft® Visual Basic® programming system. DB-Library for Visual Basic is a set of Visual Basic functions and routines that lets your application interact with Microsoft SQL Server. This book is written for programmers who are using Visual Basic programming system to develop SQL Server applications for the Microsoft Windows operating system.

To use this documentation effectively, you should already be familiar with Visual Basic and with Transact-SQL®, an enhanced version of the SQL database language. For more information about Transact-SQL, see the *Microsoft SQL Server Transact-SQL Reference*.

# Document Conventions

This documentation uses the following conventions to distinguish elements of text:

Convention	Purpose
UPPERCASE	Represents statement and clause names, structure names, datatypes, return values, filenames, and paths.
SMALL CAPITALS	Represent key names, such as CTRL.
<b>bold</b>	Represents functions, configuration-file entries, and syntax.
<i>italic</i>	Represents database names, table names, column names, index names, binding types, and variables that appear in text.
monospace	Represents examples and sample program code.

# Finding Further Information

This book contains reference information to help you write DB-Library for Visual Basic applications for Microsoft SQL Server. Information about DB-Library for Visual Basic and SQL Server 6.0 is available from the following sources.

# DB-Library Online Help

To access online Help while using DB-Library, chose the Help file icon from within Program Manager.

## SQL Server Documentation

The documentation for SQL Server 6.0 is available online as part of the Microsoft® Development Library and as Microsoft SQL Server Books Online, which is part of Microsoft SQL Server and Microsoft® SQL Server Workstation.

### *Microsoft SQL Server Setup*

Provides instructions for installing SQL Server 6.0 servers and clients.

### *Microsoft SQL Server Administrator's Companion*

Incorporates configuration, administration, and troubleshooting topics. Describes server and client configuration. Explains system administration tasks, such as managing devices and databases, backing up and recovering data, performing replication, scheduling tasks, managing security, and monitoring performance. Describes error conditions and gives troubleshooting procedures.

### *Microsoft SQL Server Database Developer's Companion*

Provides an overview of Transact-SQL, which is an enhanced version of SQL (structured query language). Discusses managing data and database objects, such as tables, views, indexes, stored procedures, defaults, rules, constraints, and user-defined datatypes.

### *Microsoft SQL Server Transact-SQL Reference*

Explains how to use Transact-SQL statements and other features to access databases, and provides syntax and descriptions for Transact-SQL statements, system procedures, utility programs, and system tables.

### *Microsoft SQL Server Programming SQL Distributed Management Objects*

Provides syntax and reference information for the SQL Distributed Management Objects, which are 32-bit OLE Automation objects for the Microsoft® Windows® 95 and Windows NT operating systems. These objects, properties, methods, and collections are used to write scripts and programs that can administer multiple SQL Servers distributed across a network.

### *Microsoft SQL Server Programming DB-Library for C*

Provides syntax and reference information for the DB-Library application programming interface (API), which is used to write C/C++ client programs for the Windows, Windows 95, Windows NT, and MS-DOS operating systems.

### *Microsoft SQL Server Programming DB-Library for Visual Basic*

The online version of this book, provides syntax and reference information for the DB-Library API, which is used to write Microsoft® Visual Basic® programming system-based client programs for the Windows operating system.

### *Microsoft SQL Server Programming Open Data Services*

Provides syntax and reference information for Open Data Services, used to write server applications for client/server systems.

# Obtaining Technical Support

Microsoft offers a variety of support options to help you get the most from Microsoft SQL Server. For information about these options, see the service and support card that came in the Microsoft SQL Server and Microsoft SQL Workstation packages, and the online support information in the Microsoft Development Library.

# Getting Started

DB-Library for Visual Basic is a set of functions and routines for building front-end applications for the Windows operating system that interact with Microsoft SQL Server. DB-Library for Visual Basic is built on and is a subset of the SQL Server client library for C programmers.

DB-Library for Visual Basic functions and routines:

- Send Transact-SQL statements to a SQL Server workgroup database.
- Process the results of Transact-SQL statements.
- Provide information about the way an application interacts with SQL Server.

For information about Transact-SQL, see the *Microsoft SQL Server Transact-SQL Reference*. For information about DB-Library for C, see *Microsoft SQL Server Programming DB-Library for C*.

With DB-Library for Visual Basic functions and routines, you can create applications that interact directly with SQL Server. DB-Library for Visual Basic applications do not require a language precompiler. As a result, your SQL Server becomes an integral part of your Visual Basic application by:

- Retrieving and updating values from a database directly
- Manipulating database values from database tables
- Inserting program values into a database
- Moving data between SQL Server and an operating-system file



# System Requirements

To develop Win16 DB-Library for Visual Basic applications, you must meet the same system requirements as a Microsoft SQL Server Windows-based client. Visual Basic for Windows version 3.0 or later is also required.

For information about the system requirements for running SQL Server, see *Microsoft SQL Server Setup*.

# DB-Library for Visual Basic Files

The following files are included with DB-Library for Visual Basic:

## In C:\SQL60\BIN

File	Description
VBSQL.VBX	Default DB-Library client library file, compatible with Microsoft Visual Basic for Windows version 3.0 or later.

## In C:\SQL60\HELP

File	Description
DBLIBVB.HLP	Help file containing reference information about DB-Library for Visual Basic functions, available for the Visual Basic for Windows programming system.

## In C:\SQL60\DBLIB\INCLUDE

File	Description
VBSQL.BAS	Include file containing all DB-Library for Visual Basic function and routine declarations.
VBSQL.BI	Old name of include file provided for backward compatibility. (Use VBSQL.BAS, not this file. The contents of this file are identical to VBSQL.BAS.)

## In C:\SQL60\DBLIB\VSAMPLES

File	Description
README.TXT	Summary of sample programs available in each directory.

# **Programming with DB-Library for Visual Basic**

This chapter discusses the concepts behind programming with DB-Library for Visual Basic. The first half of the chapter describes the fundamental building blocks of a DB-Library for Visual Basic application, and provides a framework that uses the primary set of functions around which most DB-Library for Visual Basic applications are built. The second half of the chapter summarizes the full set of DB-Library for Visual Basic functions that are available to your application.

# Application Framework

To build a DB-Library for Visual Basic application, you typically follow these steps:

1. Provide an error handler.
2. Initialize DB-Library for Visual Basic.
3. Open a connection to SQL Server.
4. Send Transact-SQL statements to SQL Server.
5. Process the results.
6. Close the connection(s) to SQL Server and exit the application.

The following sections briefly describe each of these steps, provide examples, and tell how to locate the functions described here in the sample applications for Windows.

## Error and Message Handling

When either DB-Library for Visual Basic or SQL Server encounter an error condition, the error is reported to your application. To respond to error conditions, DB-Library for Visual Basic applications must implement an error handler and a message handler.

Errors detected by DB-Library for Visual Basic (such as a failure to connect to SQL Server) invoke the error handler, and provide information such as a description of the error and a severity rating. Errors detected by SQL Server (for example, invalid SQL syntax) are sent back to the client as messages and invoke both the error handler and the message handler. Based on the error information, the code that you supply for each handler can take whatever actions you define, such as informing the user of a problem.

The sections that follow tell how to implement each type of handler, and provide examples. The sample applications include examples of error and message handlers.

# Setting Up the Error- and Message-Handling Interface

The error-handling interface between your application and DB-Library for Visual Basic is provided by a custom control. When you add this custom control to your application, it installs two error handler event procedures, assigns them the default names **VBSQL1\_Error** and **VBSQL1\_Message**, and supplies the appropriate parameter list as part of each procedure name.

You must explicitly include the DB-Library for Visual Basic procedures for handling errors in your application's primary form before you can use them in your application. These error-handling procedures are contained in the VBSQL.VBX file as part of a custom-control tool that is represented by a stop sign icon in the Toolbox palette. You can place only one error-handler control in an application.

## ▶ To include the error-handling procedures in your application

1. From the File menu, choose Add File to add the file VBSQL.VBX (available in the C:\SQL60\BIN directory by default) to your Project.

Visual Basic adds the VBSQL.VBX file to your Project window and adds the VBSQL.VBX error handler to the bottom of the Toolbox as a custom control that is shown as a stop sign.

2. Select the error-handler custom control and place it on your startup form. Only a single copy of the VBSQL.VBX control should be used for an application. The form containing the VBSQL.VBX should remain loaded as long as the DB-Library application is running, and it should be the last form unloaded when the application ends.

It appears on the form with a default name of VBSQL1. (Since the VBSQL1 custom control requires no action from the user to make it active, you can set its *Visible* property to FALSE (0) to make it invisible.) The code for the startup form now includes the object VBSQL1 and includes procedure templates for two error event handlers: **VBSQL1\_Error** and **VBSQL1\_Message**.

To implement your error handlers, add code to these event procedures to perform whatever specialized error handling is appropriate to your application. Note that you can also rename the error-handler custom control using the Properties window.

## Error- and Message-Handling Interface Syntax

This section describes the syntax for the DB-Library for Visual Basic error and message handlers. **VBSQL1\_Error** is called automatically whenever the application encounters a DB-Library for Visual Basic error. **VBSQL1\_Message** is called in response to messages returned by SQL Server.

### VBSQL1\_Error Procedure

The **VBSQL1\_Error** procedure has the following syntax:

**VBSQL1\_Error** ( *sqlconn%*, *severity%*, *errornum%*, *errorstr\$*, *retcode%* )

where

*sqlconn%*

Is a SQL Server connection that encountered an error.

*severity%*

Is an integer that denotes the severity of an error. For a list of error severities, see [Error Messages](#).

*errornum%*

Is an integer, or error number, that denotes an error constant. Error numbers and their corresponding constants are listed in the VBSQL.BAS include file. For a list of DB-Library for Visual Basic error constants and their corresponding error messages, see [Error Messages](#).

*errorstr\$*

Is a string that is the error message for a corresponding error constant. For a list of DB-Library for Visual Basic error constants and their corresponding error messages, see [Error Messages](#).

*retcode%*

Is an output parameter that determines how DB-Library for Visual Basic will proceed following the error. Custom error handling code can set *retcode%* to one of the following integers prior to exiting the **VBSQL1\_Error** procedure:

**0** INTEXIT

Treated as an INTCANCEL error.

**1** INTCONTINUE

Meaningful only for timeout errors (SQLETIME). It continues to wait for a second timeout period. At the end of the second period, the error handler is called again. In any other case, this value is considered an error and is treated as an INTCANCEL.

**2** INTCANCEL

Returns FAIL from the DB-Library for Visual Basic function that caused the error.

For timeout errors (SQLETIME) only, DB-Library will call **SqlCancel%** in an attempt to cancel the current command batch and flush any pending results. If this **SqlCancel%** attempt also times out, the connection is broken.

### VBSQL1\_Message Procedure

The **VBSQL1\_Message** procedure has the following syntax:

**VBSQL1\_Message** ( *sqlconn%*, *message&*, *state%*, *severity%*, *msgstr\$* )

where

*sqlconn%*

Is the SQL Server connection that encountered an error.

*message&*

Is an integer that is a message number returned by SQL Server. Each number has a corresponding message to which it is associated. For more information about SQL Server messages, see the *Microsoft SQL Server Administrator's Companion*.

*state%*

Is the SQL Server state number. The number provides information about the context of the current error.

*severity%*

Is an integer that denotes the SQL Server severity of an error. For information about SQL Server severities, see the *Microsoft SQL Server Administrator's Companion*.

*msgstr\$*

Is a message string corresponding to the *message* number returned by SQL Server. For more information about SQL Server messages, see the *Microsoft SQL Server Administrator's Companion*.



## Error- and Message-Handling Interface Examples

The following code fragment implements the VBSQL.VBX **VBSQL1\_Error** and **VBSQL1\_Message** procedures by calling the **UserSQLErrorHandler** and **UserSQLMsgHandler** routines defined in code module VBSQLGEN.BAS:

```
Sub VBSQL1_Error (Sqlconn%, Severity%, ErrorNum%, ErrorStr$, RetCode%)
    OsErr% = -1
    RetCode% = UserSQLErrorHandler (Sqlconn%, Severity%, ErrorNum%,
        OsErr%, ErrorStr$, OsErrStr$)
END SUB

Sub VBSQL1_Message (Sqlconn%, Message&, State%, Severity%, MsgStr$)
    UserSQLMsgHandler Sqlconn%, Message&, State%, Severity%, MsgStr$
END SUB
```

For a list of the code in the routines **UserSQLErrorHandler** and **UserSQLMsgHandler**, see the VBSQLGEN.BAS sample code.

For a list of DB-Library for Visual Basic messages, see [Error Messages](#).

## Using Declarations

Declarations for all DB-Library for Visual Basic procedures are included in the VBSQL.BAS file. You must add the contents of this file to the global declarations file for each application. For example, the QUERY sample application contains these declarations in the file QUERY.BAS.

# Initializing DB-Library for Visual Basic

The first task of a DB-Library for Visual Basic application is to call the **SqlInit\$** function to initialize DB-Library for Visual Basic. During initialization, your application becomes "registered" with DB-Library, preventing conflicts with other applications that might call this library at the same time.

Your application must call **SqlInit\$** before it calls any other DB-Library for Visual Basic function. Because no other DB-Library for Visual Basic actions can take place before the call to **SqlInit\$**, call **SqlInit\$** at the start of your application (usually the **Form\_Load** event procedure of the application's primary, or startup, form).

**SqlInit\$** takes no parameters and returns the DB-Library version number as a string (where DB-Library is the generic name for the standard SQL Server client interface on which DB-Library for Visual Basic is based). Your error handlers should be available when **SqlInit\$** is called.

The following program fragment illustrates a valid call to **SqlInit\$**:

```
'Initialize DB-Library for Visual Basic Library.  
IF SqlInit$() = "" THEN  
    PRINT "DB-Library for Visual Basic Library has not been initialized."  
    END  
END IF
```

For an example of how **SqlInit\$** is used, refer to the QUERY sample application. This sample application satisfies the initialization requirement by calling the routine **InitializeApplication**, which in turn calls **SqlInit\$**. **InitializeApplication** is defined separately in the INIEXITW.BAS common code module.

## Opening SQL Server Connections

Once the application has been initialized, your application interacts with SQL Server by opening one or more SQL Server connections with the appropriate user login information. Your application uses the connection to send Transact-SQL statements to SQL Server and to receive the results of those statements from SQL Server.

To open a SQL Server connection, use the **SqlLogin%** and **SqlOpen%** functions. DB-Library for Visual Basic also offers a function called **SqlOpenConnection%**, which provides a convenient method for logging in to SQL Server and opening a connection in one step. It combines the work of several lower-level functions into a single function, reducing the amount of code needed to log in a user and open a connection.

The following sections describe how to use **SqlLogin%** and **SqlOpen%** and provide examples for each function. For an example of how the utility function **SqlOpenConnection%** is used, refer to the QUERY sample application. This sample application logs in and opens a connection by calling the routine **LoginToServer**, which in turn calls **SqlOpenConnection%**. **LoginToServer** is defined separately in the VBSQLGEN.BAS common code module.

## Logging In to SQL Server

The **SqlLogin%** function allocates a SQL Server login record and returns a value that serves as the identifier for that login record. The login record is made up of a set of attributes that are initially assigned default values. Although you cannot manipulate the login record directly, you can use DB-Library for Visual Basic functions to specify a username, a password, an application name, and several optional attributes. For example:

- The **SqlSetLUser%** and **SqlSetLPwd%** functions set the username and password that DB-Library for Visual Basic will use to log in to SQL Server.
- The **SqlSetLApp%** function sets the application name that appears in the SQL Server *sysprocesses* system table.

A Visual Basic form is a good tool for obtaining the required login information from the user. The following program fragment uses **SqlLogin%** to allocate a login record:

```
'Get a login record and set login attributes.  
Login% = SqlLogin%()  
Result% = SqlSetLUser%(Login%, "loginid")  
Result% = SqlSetLPwd%(Login%, "passwd")  
Result% = SqlSetLApp%(Login%, "example")
```

## Opening a Connection

The **SqlOpen%** function logs in to SQL Server (using the **SqlLogin%** login record) and establishes a connection. The parameters required by the **SqlOpen%** function are the **SqlLogin%** login record identifier and a SQL Server server name. For each connection it establishes, **SqlOpen%** returns a connection identifier.

The **SqlOpen%** connection identifier remains associated with the connection as long as the connection is active. This documentation uses the variable *sqlconn%* to represent the connection identifier returned by **SqlOpen%**. Because most DB-Library for Visual Basic functions perform operations that are associated with a particular connection, most functions require you to specify the *sqlconn%* connection identifier as the first parameter.

**Important** Do not modify a connection's identifier in any way. If you modify an identifier, the connection and its associated data can be lost.

Once you open a connection with the **SqlOpen%** function, a data structure is associated with that connection to store a variety of information about the connection and its interaction with SQL Server. Your application uses DB-Library for Visual Basic functions to extract this information.

The following program fragment uses **SqlOpen%** to open a single SQL Server connection:

```
'Get a connection for communicating with SQL Server.  
Sqlconn% = SqlOpen%(Login%, "server")
```

You can use **SqlOpen%** to open two or more SQL Server connections at the same time.

## Sending Transact-SQL Statements to SQL Server

The **SqlCmd%** and **SqlExec%** functions allow your application to execute Transact-SQL statements.

The **SqlCmd%** function fills the command buffer with Transact-SQL statements that you then send to SQL Server. Each succeeding call to **SqlCmd%** appends the supplied Transact-SQL text to the end of text already in the buffer. When you add text to text already in the buffer, be sure to supply necessary blanks between words, such as the blank space at the beginning of the WHERE clause in the program fragment that follows.

```
'Retrieve two columns from the "authors" table
'in the "pubs" database.

'Put the command into the command buffer.
Result% = SqlCmd%(Sqlconn%, "SELECT au_lname, city")
Result% = SqlCmd%(Sqlconn%, " FROM pubs..authors")
Result% = SqlCmd%(Sqlconn%, " WHERE state = 'CA'")

'Send the command to SQL Server and start execution.
Result% = SqlExec%(Sqlconn%)
```

**SqlExec%** sends the contents of the buffer to SQL Server, which parses and executes the specified Transact-SQL statements. You can also use **SqlSend%** and **SqlOk%** instead of **SqlExec%**.

For an example of how these functions are used, refer to the QUERY sample application. This sample application calls the routine **ExecuteSqlCommand**, which in turn calls the functions **SqlCmd%** and **SqlExec%**. **ExecuteSqlCommand** is defined separately in the VBSQLGEN.BAS common code module.

# Processing Results

The functions **SqlResults%**, **SqlNextRow%**, and **SqlData\$** or **SqlIData\$** process the results of Transact-SQL statements and return information to the user.

The **SqlResults%** function sets up the results of the current Transact-SQL statement for processing. Normally, you call **SqlResults%** once for every Transact-SQL statement placed in the command buffer (**SqlResults%** should return SUCCEED), and then one last time to return the NOMORERESULTS indicator. In the preceding program fragment, three calls to **SqlCmd%** are used to place a single Transact-SQL statement in the command buffer. In this case, you need to call **SqlResults%** only twice. The first call should return SUCCEED, and the second call should return NOMORERESULTS.

When a call to **SqlResults%** returns SUCCEED (indicating that the results of the current Transact-SQL statement are available), call **SqlNextRow%** to read a row of data from SQL Server. Each successive call to **SqlNextRow%** reads another row until the last row has been read and the NOMOREROWS indicator is returned. Row processing must take place after **SqlNextRow%** returns SUCCEED and before the next call to **SqlNextRow%**, because each call to **SqlNextRow%** overwrites the values in the previous row (unless row buffering has been turned on). You must call **SqlNextRow%** until it returns NOMOREROWS (to complete the processing of a result set) before calling **SqlResults%** again. The following program fragment uses a DO loop to call **SqlNextRow%** until NOMOREROWS is returned.

```
Result% = SqlResults%(Sqlconn%)

'Process the command.
IF Result% = SUCCEED THEN

    'Retrieve and print the result rows.
    PRINT
    DO UNTIL SqlNextRow%(Sqlconn%) = NOMOREROWS
        Name$ = SqlData$(Sqlconn%, 1)
        City$ = SqlData$(Sqlconn%, 2)
        PRINT Name$, City$
    LOOP

END IF
```

Even when you know that a Transact-SQL query returns only one row, you must call **SqlNextRow%** twice: once to get the result row, and once to get the NOMOREROWS flag.

Transact-SQL statements return two types of result rows: regular rows and compute rows. Regular rows are generated from columns in a Transact-SQL SELECT statement. Compute rows are generated from columns in a COMPUTE clause. Because these two types of rows contain very different data, the application must process them separately.

The **SqlData\$** function returns a string containing data from a regular result column. The **SqlIData\$** function returns a string containing data from a compute column. This data is always returned as a string, regardless of its datatype. Binary strings of the datatypes *binary*, *varbinary*, and *image* are returned as binary strings.

For an example of how these functions are used, refer to the QUERY sample application. This sample application calls the routines **GetDatabases** and **Process\_Sql\_query**, defined separately in the VBSQGEN.BAS common code module.



## Closing SQL Server Connections

The **SqlExit** routine closes all SQL Server connections that have been opened by your application and frees the system memory associated with each connection. The following program fragment illustrates the use of **SqlExit**:

```
'Close connections.  
SqlExit  
END
```

To close a single specified SQL Server connection, use **SqlClose**.

To free the additional memory associated with a Windows-based application, you must make an additional call to the **SqlWinExit** function, described in [Exiting a Windows-based Application](#).

For an example of how **SqlExit** is used, refer to the QUERY sample application. This sample application calls the routine **ExitApplication**, which in turn calls **SqlExit**. **ExitApplication** is defined separately in the INIEXITW.BAS common code modules.

## Exiting a Windows-based Application

For Windows, your application must make an additional call to free all memory associated with your application. More than one application can reference DB-Library for Visual Basic at the same time. To avoid conflicts between these applications, DB-Library for Visual Basic allocates memory to store information identifying each application. When you exit a DB-Library for Visual Basic application, you must call **SqlWinExit** to release this memory and make it available to other applications.

The following program fragment illustrates the use of **SqlWinExit**:

```
SUB Quit_Click()  
    'Close and free all SQL Server connections.  
    SqlExit  
  
    'Release memory allocated for VBSQL.  
    SqlWinExit  
  
    'End VBSQL application.  
    END  
END SUB
```

Typically, Windows-based applications include an Exit or Quit menu option with the File menu. The Visual Basic event procedure associated with the Exit menu option must include calls to both **SqlExit** and **SqlWinExit** to close all connections and release memory allocated to the application.

## Example

This example program provides the framework for most DB-Library for Visual Basic applications. The example uses sample subroutine names for Visual Basic event procedures with which this code might be associated.

```
SUB Form_Load()
    'Initialize VBSQL.
    IF SqlInit$() = "" THEN
        PRINT "VBSQL has not been initialized."
    END
END IF
END SUB

SUB LoginCmd_Click()
    'Get a Login record and set login attributes.
    Login% = SqlLogin%()
    loginid = LoginIDText.Text
    passwd = PasswordText.Text
    example = ExampleText.Text
    Result% = SqlSetLUser%(Login%, loginid)
    Result% = SqlSetLPwd%(Login%, passwd)
    Result% = SqlSetLApp%(Login%, example)

    'Get a connection for communicating with SQL Server.
    server = ServerText.Text
    Sqlconn% = SqlOpen%(Login%, server)
END SUB

SUB ExecuteCmd_Click()
    'Retrieve two columns from the "authors" table
    'in the "pubs" database.

    'Put the command into the command buffer.
    Result% = SqlCmd%(Sqlconn%, "SELECT au_lname, city")
    Result% = SqlCmd%(Sqlconn%, " FROM pubs..authors")
    Result% = SqlCmd%(Sqlconn%, " WHERE state = 'CA'")

    'Send the command to SQL Server and start execution.
    Result% = SqlExec%(Sqlconn%)
    Result% = SqlResults%(Sqlconn%)

    'Process the command.
    IF Result% = SUCCEED THEN
        'Retrieve and print the result rows.
        PRINT
        DO UNTIL SqlNextRow%(Sqlconn%) = NOMOREROWS
            Name$ = SqlData$(Sqlconn%, 1)
            City$ = SqlData$(Sqlconn%, 2)
            PRINT Name$, City$
        LOOP
    END IF
END SUB
```

```
SUB QuitApp_Click()  
    'Close connection and exit program.  
    SqlExit  
    SqlWinExit  
    END  
END SUB  
  
Sub VBSQL1_Error (Sqlconn%, Severity%, ErrorNum%, ErrorStr$, RetCode%)  
    MsgBox ("DB-Library Error: " + Str$(ErrorNum%) + " " + ErrorStr$)  
END SUB  
  
Sub VBSQL1_Message (Sqlconn%, Message&, State%, Severity%, MsgStr$)  
    MsgBox ("SQL Server Error: " + Str$(Message&) + " " + MsgStr$)  
END SUB
```

# Advanced Topics

This section includes useful information about specific groups of DB-Library for Visual Basic functions.

## Retrieving Regular Result Column Information

Call the following DB-Library for Visual Basic functions to retrieve information on regular result columns after **SqlExec%** or **SqlOk%** returns SUCCEED.

Function	Description
<b>SqlColLen%</b>	Returns the maximum length of the data, in bytes, converted to a string in a regular column.
<b>SqlColName\$</b>	Returns the name of a regular result column.
<b>SqlColType%</b>	Returns the SQL Server datatype for a regular result column.
<b>SqlDatLen&amp;</b>	Returns the actual length, in bytes, converted to a string of the data in a regular column. This function is often used in conjunction with <b>SqlData\$</b> . The value returned by <b>SqlDatLen&amp;</b> can be different for each row that <b>SqlNextRow%</b> reads.
<b>SqlNumCols%</b>	Returns the number of columns in the current set of results.
<b>SqlPrType\$</b>	Converts a SQL Server token value to a string.

## Retrieving Compute Result Column Information

Call the following DB-Library for Visual Basic functions to retrieve information on compute result columns after **SqlNextRow%** or **SqlGetRow%** returns a compute ID.

Function	Description
<b>SqlADLen&amp;</b>	Returns the actual length of the data in a compute column in the form of a string. This function is often used in conjunction with <b>SqlADData\$</b> . The value returned by <b>SqlADLen&amp;</b> can be different for each compute row that <b>SqlNextRow%</b> reads.
<b>SqlAltColId%</b>	Returns the identification number in a compute column.
<b>SqlAltLen%</b>	Returns the maximum length of the data, in bytes. The length is converted to a string in a compute column.
<b>SqlAltOp%</b>	Returns the type of aggregate function of a compute column.
<b>SqlAltType%</b>	Returns the datatype of a compute column.
<b>SqlByList\$</b>	Returns the bylist of a compute row.
<b>SqlNumAlts%</b>	Returns the number of columns in a compute row.
<b>SqlNumCompute%</b>	Returns the number of COMPUTE clauses in the current set of results.

# Browse Mode

With DB-Library for Visual Basic browse mode, your application can browse through database rows and update their values one row at a time. Each row must be transferred from the database into program variables before it can be browsed and updated. As a result, browsing involves several steps.

Because a row being browsed is not the actual row residing in the database but a copy residing in program variables, changes to the values in those variables must be transferred reliably to ensure that the original database row is kept up to date. In particular in multiuser situations, updates made to the database by one user must not incorrectly overwrite updates made by another user. Regulating updating is especially important because an application typically selects a number of rows from the database at once, but users browse and update the database one row at a time. A timestamp column in browsable database tables provides the information necessary to regulate multiuser updating.

Browse-mode functions also allow an application to handle ad hoc queries. Several browse-mode functions return information that an application can use to examine the structure of a complicated ad hoc query.

## ▶ To implement the DB-Library for Visual Basic browse mode in an application

1. Execute a SELECT statement, generating result rows containing result columns. The SELECT statement must include the FOR BROWSE option.
2. Copy the values in the result column into the program variables one row at a time.
3. If appropriate, change the variables' values in response to user input.
4. If appropriate, execute an UPDATE statement that updates the database row corresponding to the current result row. To handle multiuser updates, the WHERE clause of the UPDATE statement must refer to the timestamp column. You can obtain a WHERE clause with the **SqlQual\$** function.
5. Repeat steps 2 through 4 for each result row.

To use browse mode, the following conditions must be true:

- The SELECT statement must end with the keywords FOR BROWSE.
- The table(s) to be updated must have a unique index and timestamp column.
- The result columns used in the updates must derive from browsable tables and cannot be the result of SQL expressions, such as MAX(*colname*). In other words, there must be a valid correspondence between the result column and the database column to be updated.

In addition to these conditions, the browse mode always requires two connections – one for selecting the data and another for performing updates based on the selected data.



# Building Applications

This chapter discusses building a DB-Library for Visual Basic application. The guidelines it describes apply to the development of DB-Library for Visual Basic applications.

# Overview

Developing a Visual Basic application involves two basic processes:

- Providing a user interface that requests information from and conveys information to the person using the application.
- Supplying the programming code that performs a set of tasks in response to specific user actions.

These two processes are strongly interdependent. When the user of the application uses a particular screen object to initiate an action, Visual Basic responds by invoking the Basic code contained in whatever event procedure is associated with that screen object. User interaction with SQL Server occurs when the Basic code for an event procedure contains functions and routines from DB-Library for Visual Basic.

Unless your DB-Library for Visual Basic application is very simple, use multiple forms to divide an application into logical parts. For example, a form that gets login information from a user ties together the user data and the functions that act on that user data. The login information and the login functions are part of the same process – logging in to SQL Server.

DB-Library for Visual Basic applications can be large and complex, and can manage several SQL Server processes. For example, an application can manage querying databases, processing and retrieving information, accessing stored procedures, bulk-copying data, and so on. Dividing applications into forms and modules that control the various processes makes it easier to maintain and change the applications.

# Required Project Files

When testing or building a DB-Library for Visual Basic application, the Visual Basic Project that contains your application's form and module files must use DB-Library for Visual Basic files described in the sections below.

For DB-Library for Visual Basic applications, add the files VBSQL.VBX and VBSQL.BAS to each Visual Basic application Project.

- The VBSQL.VBX file provides DB-Library for Visual Basic of functions and routines.
- The VBSQL.BAS file contains all the DB-Library for Visual Basic function and routine declarations and global constants.

When testing or building a DB-Library for Visual Basic application, all the information in the VBSQL.BAS file must be included in a global module for your Project. You can either make VBSQL.BAS the global module for your Project, or you can copy the contents of the VBSQL.BAS file and paste it into an existing global module.

# Setting Up the Development Environment

The Visual Basic development environment allows you to test your application interactively during the development process.

Before you start interactive development of SQL Server applications in Windows, verify that your PATH command includes the DB-Library client support file, MSDBLIB3.DLL, and a corresponding Net-Library file, such as DBNMP3.DLL. These files, available with your SQL Server client software for Windows, allow you to test your connection to SQL Server using either custom application functions or a Windows-based tool such as the ISQL/w utility.

A new Visual Basic Project that uses DB-Library for Visual Basic must include the VBSQL.VBX custom control as a Project file. Adding this file includes the custom error-handling control (stop sign icon) to your Toolbox palette. Be sure to place one instance of this custom control on the startup form of your application.

# Running Applications

To run your completed executable application, you must first make sure that the SQL Server communications software is loaded at the client computer.

For Windows-based applications, the client computer must have the VBSQL.VBX and MSDBLIB3.DLL files, and an appropriate Net-Library such as DBNMP3.DLL.

# Core Functions

[SqlADData\\$](#)

[SqlADLen&](#)

[SqlAltCollId%](#)

[SqlAltLen%](#)

[SqlAltOp%](#)

[SqlAltType%](#)

[SqlAltUType&](#)

[SqlByList\\$](#)

[SqlCancel%](#)

[SqlCanQuery%](#)

[SqlChange\\$](#)

[SqlClose](#)

[SqlClrBuf](#)

[SqlClrOpt%](#)

[SqlCmd%](#)

[SqlCmdRow%](#)

[SqlCollInfo%](#)

[SqlCollLen%](#)

[SqlColName\\$](#)

[SqlColType%](#)

[SqlColUType&](#)

[SqlCount&](#)

[SqlCurCmd%](#)

[SqlCurRow&](#)

[SqlData\\$](#)

[SqlDataReady%](#)

[SqlDateCrack%](#)

[SqlDatLen&](#)

[SqlDead%](#)

[SqlExit](#)

[SqlFirstRow&](#)

[SqlFreeBuf](#)

[SqlFreeLogin](#)

[SqlGetChar\\$](#)

[SqlGetMaxProcs%](#)

[SqlGetOff%](#)  
[SqlGetPacket%](#)  
[SqlGetRow%](#)  
[SqlGetTime%](#)  
[SqlInit\\$](#)  
[SqlIsAvail%](#)  
[SqlIsCount%](#)  
[SqlIsOpt%](#)  
[SqlLastRow&](#)  
[SqlLogin%](#)  
[SqlMoreCmds%](#)  
[SqlName\\$](#)  
[SqlNextRow%](#)  
[SqlNumAlts%](#)  
[SqlNumCols%](#)  
[SqlNumCompute%](#)  
[SqlNumOrders%](#)  
[SqlOpen%](#)  
[SqlOrderCol%](#)  
[SqlProcInfo%](#)  
[SqlPrType\\$](#)  
[SqlResults%](#)  
[SqlRows%](#)  
[SqlRowType%](#)  
[SqlServerEnum%](#)  
[SqlSetAvail](#)  
[SqlSetLApp%](#)  
[SqlSetLHost%](#)  
[SqlSetLNatLang%](#)  
[SqlSetLoginTime%](#)  
[SqlSetLPacket%](#)  
[SqlSetLPwd%](#)  
[SqlSetLSecure%](#)  
[SqlSetLUser%](#)  
[SqlSetLVersion%](#)  
[SqlSetMaxProcs%](#)  
[SqlSetOpt%](#)

SqlSetTime%

SqlExec%

SqlOk%

SqlSend%

SqlStrCpy%

SqlStrLen%

SqlUse%

SqlWinExit



# SqlADData\$

Returns the data in a COMPUTE clause column.

## Syntax

**SqlADData\$** ( *sqlconn%*, *computeid%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*computeid%*

Is the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, which can have varying numbers of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

*column%*

Is the number of the column. The first column returned is number 1.

## Returns

A string containing the data in the compute column. For the SQL Server datatypes *binary*, *varbinary*, and *image*, **SqlADData\$** returns a string of binary data. For all other datatypes, **SqlADData\$** returns a string of characters. When there is no such column or when the data is NULL, an empty string is returned. To make sure that the data really is NULL, always check for a return of 0 by using **SqlADLen&**.

## Remarks

After each call to **SqlNextRow%** that returns a value greater than 0, use **SqlADData\$** to obtain the data in a particular COMPUTE clause column. The data is not null-terminated. Use **SqlADLen&** to get the length of the data in a COMPUTE clause column.

## Example

```
'Put commands into the command buffer.
cmd$ = "SELECT db_name(dbid), dbid, size FROM sysusages"
cmd$ = cmd$ + " ORDER BY dbid"
cmd$ = cmd$ + " COMPUTE SUM(size) BY dbid"
Result% = SqlCmd$(Sqlconn%, cmd$)
```

```
'Send commands to SQL Server and start execution.
Result% = SqlExec$(Sqlconn%)
Result% = SqlResults$(Sqlconn%)
```

```
'Examine the results of the COMPUTE clause.
IF Result% = SUCCEED THEN
    DO UNTIL Result% = NOMOREROWS
        Result% = SqlNextRow$(Sqlconn%)
        IF Result% = NOMOREROWS THEN Exit DO
        IF Result = REGROW THEN
            PRINT "regular row returned."
            PRINT
        ELSE
            'This row is the result of a COMPUTE clause
            'and Result% is the computeid% of this
            'COMPUTE clause
```

```
        sum = Val(SqlADData$(Sqlconn%, Result%, 1))
        PRINT "sum = "; sum
    PRINT
END IF
LOOP
```

**See Also**

[SqlADLen&](#), [SqlAltLen%](#), [SqlAltType%](#), [SqlGetRow%](#), [SqlNextRow%](#), [SqlNumAlts%](#)

# SqlADLen&

Returns the length of the data in a compute column.

## Syntax

**SqlADLen&** ( *sqlconn%*, *computeid%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*computeid%*

Is the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, which can have varying numbers of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

*column%*

Is the column number. The first column is number 1.

## Returns

The length, in bytes, of the data in a compute column. When no such column or COMPUTE clause exists, -1 is returned. When the data has a null value, 0 is returned.

## Remarks

### Example

'Put the command into the command buffer.

```
cmd$ = "SELECT name FROM sysobjects"
```

```
cmd$ = cmd$ + " ORDER BY name"
```

```
cmd$ = cmd$ + " COMPUTE MAX(name) "
```

'Send the command to SQL Server and start execution.

```
Result% = SqlCmd$(Sqlconn%,cmd$)
```

```
Result% = SqlExec$(Sqlconn%)
```

'Process the results of each statement.

```
DO UNTIL Result% = NOMOREROWS
```

```
    Result% = SqlNextRow$(Sqlconn%)
```

```
    IF Result% = NOMOREROWS THEN Exit DO
```

```
    ELSE IF Result = REGROW THEN
```

```
        PRINT "regular row returned."
```

```
        PRINT
```

```
    ELSE
```

```
        'This row is the result of a COMPUTE clause.
```

```
        DataLength& = SqlADLen&(Sqlconn%, ComputeID%, 1)
```

```
        Data$ = SqlAData$(Sqlconn%, ComputeID%, 1)
```

```
        PRINT "The length of " + Data$ + " is " + DataLength& + "bytes."
```

```
    END IF LOOP
```

## See Also

[SqlAData\\$, SqlAltLen%, SqlAltType%, SqlGetRow%, SqlNextRow%, SqlNumAlts%](#)

# SqlAltColId%

Returns the ID of a compute column.

## Syntax

**SqlAltColId%** ( *sqlconn%*, *computeid%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*computeid%*

Is the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, which can have varying numbers of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

*column%*

Is the number of the compute column. The first column returned is number 1.

## Returns

The ID that the aggregate operator in the COMPUTE clause column applies to. The first column is number 1. When either the *computeid%* or the *column%* is invalid, -1 is returned. Call this function after **SqlNextRow%** or **SqlGetRow%** returns a *computeid*.

## Remarks

**SqlAltColId%** returns the ID of the column to which the aggregate operator applies for a particular column in a COMPUTE clause. For example, the function **SqlAltColId%(sqlconn%, 1, 1)** returns 2, since the COMPUTE COUNT clause in the following example refers to the second column in the select list:

```
select dept, name from employee
order by dept, name
compute count(name) by dept
```

## See Also

[SqlADData\\$, SqlADLen&, SqlAltLen%, SqlGetRow%, SqlNextRow%, SqlNumAlts%, SqlPrType\\$](#)

# SqlAltLen%

Returns the maximum length of the data in a compute column.

## Syntax

**SqlAltLen%** ( *sqlconn%*, *computeid%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*computeid%*

Is the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, which can have varying numbers of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

*column%*

Is the number of the column. The first column returned is number 1.

## Returns

The maximum number of bytes that the data can be in a particular column in a COMPUTE clause. When no such column or COMPUTE clause exists, -1 is returned.

## Remarks

## Example

In the following example, **SqlAltLen%(sqlconn%, 1, 1)** returns 11 because COUNT is an *int* datatype, which is 11 bytes long:

```
select dept, name from employee
order by dept, name
compute count(name) by dept
```

## See Also

[SqlADData\\$, SqlADLen&, SqlAltType%, SqlGetRow%, SqlNextRow%, SqlNumAlts%](#)

# SqlAltOp%

Returns the type of aggregate function in a compute column.

## Syntax

**SqlAltOp%** ( *sqlconn%*, *computeid%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*computeid%*

Is the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, which can have varying numbers of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

*column%*

Is the number of the column. The first column returned is number 1.

## Returns

The type of aggregate function in a particular column in a COMPUTE clause, as follows:

Aggregate function type	Aggregate operator
SQLAOPSUM	SUM
SQLAOPAVG	AVG
SQLAOPCNT	COUNT
SQLAOPMIN	MIN
SQLAOPMAX	MAX

Call **SqlAltOp%** only after **SqlNextRow%** or **SqlGetRow%** returns a *computeid*. When the *computeid%* or *column%* is not valid, -1 is returned.

## Remarks

**SqlAltOp%** returns the type of aggregate function in a particular column in a COMPUTE clause. In the following example, **SqlAltOp%(sqlconn%, 1, 1)** returns the type for COUNT because the first aggregate operator in the first COMPUTE clause is COUNT:

```
select dept, name from employee
order by dept, name
compute count(name) by dept
```

To convert the type to a readable string, use **SqlPrType\$**.

## See Also

[SqlADData\\$](#), [SqlADLen&](#), [SqlAltLen%](#), [SqlGetRow%](#), [SqlNextRow%](#), [SqlNumAlts%](#), [SqlPrType\\$](#)

# SqlAltType%

Returns the SQL Server datatype of a compute column.

## Syntax

**SqlAltType%** ( *sqlconn%*, *computeid%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*computeid%*

Is the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, which can have varying numbers of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

*column%*

Is the number of the column. The first column returned is number 1.

## Returns

The type of data in a column in the COMPUTE clause, as follows:

Column datatype	Returned constant
<i>char</i>	SQLCHAR
<i>varchar</i>	SQLCHAR
<i>binary</i>	SQLBINARY
<i>varbinary</i>	SQLBINARY
<i>tinyint</i>	SQLINT1
<i>smallint</i>	SQLINT2
<i>int</i>	SQLINT4
<i>real</i>	SQLFLT4
<i>float</i>	SQLFLT8
<i>smallmoney</i>	SQLMONEY4
<i>money</i>	SQLMONEY
<i>decimal</i>	SQLDECIMAL
<i>numeric</i>	SQLNUMERIC
<i>smalldatetime</i>	SQLDATETIME4
<i>datetime</i>	SQLDATETIME
<i>image</i>	SQLIMAGE
<i>text</i>	SQLTEXT

Call **SqlAltType%** only after **SqlNextRow%** or **SqlGetRow%** returns a *computeid*.

If either the *computeid%* or the *column%* is invalid, - 1 is returned.

## Remarks

**SqlAltType%** returns the datatype in a particular column in a COMPUTE clause. In the following example, **SqlAltType%(sqlconn%, 1, 1)** returns the type for SQLINT4 because the counts are of *sqlint4* datatype.

```
select dept, name from employee  
order by dept, name  
compute count(name) by dept
```

To convert the type to a readable string, use **SqIPrType\$**.

#### **See Also**

[SqIData\\$, SqADLen&, SqAltLen%, SqGetRow%, SqNextRow%, SqNumAlts%, SqIPrType\\$;](#)  
[DB-Library for Visual Basic Options.](#)



# SqlAltUType&

Returns the user-defined datatype of a compute column.

## Syntax

**SqlAltUType&** ( *sqlconn%*, *computeid%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*computeid%*

Is the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, which can have varying numbers of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

*column%*

Is the number of the column. The first column returned is number 1.

## Returns

The user-defined datatype of the specified compute column on success, or - 1 on error.

## Remarks

Call **SqlAltUType&** only after **SqlNextRow%** or **SqlGetRow%** returns a compute ID.

## See Also

[SqlAData\\$](#), [SqlGetRow%](#), [SqlNextRow%](#), [SqlNumAlts%](#)

# SqlByList\$

Returns the bylist for a COMPUTE row.

## Syntax

**SqlByList\$** ( *sqlconn%*, *computeid%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*computeid%*

Is the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, which can have varying numbers of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

*column%*

Is the number of the column. The first column is number 1.

## Returns

A binary string of column positions for the bylist in the SELECT statement of the current SQL commands.

## Remarks

A COMPUTE clause in a SELECT statement can contain the keyword BY, followed by a list of columns. This list, known as the bylist, divides the results of the COMPUTE clause into subgroups based on changing values in the specified columns. The COMPUTE clause's row aggregate is applied to each subgroup, generating a COMPUTE clause row for each subgroup.

The number of columns is equal to the length of the binary string and can be determined by calling **LEN( )** with the returned string. Each character is a binary value representing the position of a column in the bylist.

Call **SqlByList\$** after **SqlResults%** returns SUCCEED.

## Example

The following program fragment shows how to convert a binary value to an integer to get the number of a column position:

```
ByList$ = SqlByList$(Sqlconn%, ComputeID%)
DIM ByListNumber(LEN(ByList$)) as Integer
FOR x% = 1 to LEN(ByList$)
    ByListNumber(x%) = ASC(MID$(ByList$,x%,1))
NEXT x%
```

## See Also

[SqlAData\\$, SqlADLen&, SqlAltLen%, SqlAltType%, SqlColName\\$, SqlGetRow%, SqlNextRow%, SqlResults%](#)

# SqlCancel%

Cancels the execution of the statements in the command buffer and flushes any pending results.

## Syntax

**SqlCancel%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

When you call **SqlCancel%**, SQL Server stops executing the statements associated with *sqlconn%* or *sqlconn&* in the command buffer. Any pending results are read and discarded. You can call **SqlCancel%** after calling **SqlExec%**, **SqlSend%**, **SqlOk%**, **SqlResults%**, or **SqlNextRow%**. **SqlCancel%** cancels all statements in the command buffer. To cancel a single statement in a command buffer containing multiple statements, use **SqlCanQuery%**.

If you receive the DB-Library error 10038 "Results Pending," you can call **SqlCancel** to clear the pending results. If **SqlCancel** returns FAIL, the server may not be able to respond to the cancel request. Either continue to call **SqlCancel** until it returns SUCCEED, or close the DBPROCESS connection and open a new one.

## See Also

[SqlCanQuery%](#), [SqlExec%](#), [SqlNextRow%](#), [SqlOk%](#), [SqlResults%](#), [SqlSend%](#)

# SqlCanQuery%

Cancels any rows pending from the most recently executed query.

## Syntax

**SqlCanQuery%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

Use **SqlCanQuery%** to cancel any unread rows that result from the most recently executed SQL query. Calling **SqlCanQuery%** is the same as calling **SqlNextRow%** until it returns NOMOREROWS.

The **SqlResults%** function must return SUCCEED before an application can call **SqlCanQuery%**.

To ignore the results of all of the statements in the command buffer, use **SqlCancel%**.

## See Also

[SqlCancel%](#), [SqlNextRow%](#), [SqlResults%](#), [SqlSend%](#)

# SqlChange\$

Determines whether a command batch has changed the current database to another database.

## Syntax

**SqlChange\$** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The name of the new database, if any. If the database has not changed, an empty string is returned.

## Remarks

**SqlChange\$** informs the application of a switch from one database to another by catching any instance of the Transact-SQL USE statement.

When connected to SQL Server 4.2, a USE statement does not take effect until the end of the batch. The **SqlChange\$** function is therefore useful only in determining whether the current command batch has changed to another database for subsequent command batches. The simplest way to keep track of database switches is to call **SqlChange\$** when **SqlResults%** returns NOMORERESULTS at the end of each command batch.

When connected to SQL Server 6.0, a USE statement takes effect immediately.

You can also get the name of the current database by calling **SqlName\$**.

## See Also

[SqlExec%](#), [SqlName\\$](#), [SqlResults%](#), [SqlSend%](#), [SqlUse%](#)

# SqlClose

Closes and frees a single SQL Server connection.

## Syntax

**SqlClose** ( *sqlconn%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED(1) or FAIL(0).

## Remarks

**SqlClose** stops any activity associated with a SQL Server connection, closes the connection, and frees allocated memory. Closing a *sqlconn* connection with **SqlClose** automatically closes all the cursors associated with it.

To open a SQL Server connection, use **SqlOpen%**. To close all open SQL Server connections, use **SqlExit**.

Calling **SqlClose** with a *sqlconn%* or *sqlconn&* value not returned by **SqlOpen%** causes an error.

## See Also

[SqlExit](#), [SqlOpen%](#)

# SqlClrBuf

Clears rows from the row buffer.

## Syntax

**SqlClrBuf** ( *sqlconn%*, *rows&* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*rows&*

Is the number of rows to clear from the row buffer. If *rows&* is equal to or greater than the number of rows in the buffer, all but the newest row are removed. If *rows&* is less than 1, the call is ignored.

Rows are cleared on a first-in/first-out basis.

## Remarks

DB-Library for Visual Basic buffers rows as they are returned from SQL Server. If the row buffer fills completely, you can use **SqlClrBuf** to drop rows from the buffer.

Turn on row buffering by using **SqlSetOpt%** with the SQLBUFFER option as follows:

**SqlSetOpt%**(*sqlconn%*, **SqlBUFFER**, "*n*")

Replace *n* with the number of rows you want to buffer. After turning buffering on, you can randomly refer to rows that have been read from SQL Server by using **SqlGetRow%**. The row buffer can fill completely when SQL Server returns more than the number of rows you specify. When the row buffer is full, **SqlNextRow%** returns BUFFULL until you free at least one row by calling **SqlClrBuf**. **SqlClrBuf** frees the oldest rows in the buffer.

## See Also

[SqlGetRow%](#), [SqlNextRow%](#), [SqlSetOpt%](#); [DB-Library for Visual Basic Options](#)

# SqlClrOpt%

Clears an option set by **SqlSetOpt%**.

## Syntax

**SqlClrOpt%** ( *sqlconn%*, *opt%*, *optparam\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**. When *sqlconn%* is 0, the specified option is cleared for every active SQL Server connection.

*opt%*

Is the option to be cleared. Do not enclose *opt%* in quotation marks. For a list of the options available in DB-Library for Visual Basic, see [DB-Library for Visual Basic Options](#).

*optparam\$*

Is a parameter for an option. You must include *optparam\$* for all options, whether or not they take parameters. Enclose *optparam\$* in quotation marks.

If an option does not take a parameter, *optparam\$* is ignored. If an option does take a parameter, *optparam\$* is ignored for all options except SQLOFFSET and SQLSTAT.

The SQLOFFSET and SQLSTAT options can have several settings, each with a different parameter. In these cases, **SqlClrOpt%** needs a valid *optparam\$* to determine which option parameter to clear.

For a list of the options available in DB-Library for Visual Basic, see [DB-Library for Visual Basic Options](#).

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlClrOpt%** turns off options that have been set with **SqlSetOpt%**. Although you can set and clear some options directly through Transact-SQL, design your application to use **SqlSetOpt%** and **SqlClrOpt%** to set and clear options because **SqlSetOpt%** and **SqlClrOpt%** provide a uniform interface for setting and clearing all options. With these functions, your application can also use **SqlIsOpt%** to check the status of an option.

**SqlClrOpt%** does not immediately clear the options specified (with the exception of SQLBUFFER and SQLNOAUTOFREE). Options are cleared when you send the statements in the command buffer to SQL Server by invoking **SqlExec%**. Also, the results of the command generated by **SqlClrOpt%** are not returned until the command is transferred to SQL Server. Therefore, design your application to expect the results returned from the command string generated by **SqlClrOpt%**. If an invalid parameter is specified, it is not detected until the command is sent to SQL Server and the results for that command are returned using **SqlResults%**.

Note that the command string generated by this function is not immediately sent to SQL Server. Instead, it is buffered within DB-Library and sent the next time **SqlExec%** is invoked. Therefore, any options requested by this function do not go into effect until then. Also, the results of the command generated by this function are not returned until the command is transferred to SQL Server. The application should be expecting the results returned from the command string generated by this function.

For a list of the options available in DB-Library for Visual Basic, see [DB-Library for Visual Basic Options](#).

## See Also



**SqlIsOpt%**, **SqlSetOpt%**; DB-Library for Visual Basic Options

# SqlCmd%

Adds Transact-SQL text to the command buffer.

## Syntax

**SqlCmd%** ( *sqlconn%*, *cmd\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*cmd\$*

Is a character string to be copied into the command buffer.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlCmd%** adds text to the existing command buffer. The added text does not delete or overwrite the current contents until the contents of the buffer are sent to SQL Server.

After a call to **SqlExec%** or **SqlSend%**, the first call to **SqlCmd%** automatically clears the command buffer before the new text is entered. If you do not want the command buffer automatically cleared, set the SQLNOAUTOFREE option with **SqlSetOpt%**. When SQLNOAUTOFREE is set, the command buffer is cleared only by a call to **SqlFreeBuf**.

The size of the command buffer depends on the amount of available memory.

**Note** You can call **SqlCmd%** repeatedly. Sequential calls are concatenated, so be sure to add any necessary blanks at the end of one line or the beginning of the next.

## Example

The following code fragment shows how to use **SqlCmd%** to build up a multiline SQL command:

```
Result% = SqlCmd%(Sqlconn%, "SELECT name FROM sysobjects")
Result% = SqlCmd%(Sqlconn%, " WHERE id = 5")
Result% = SqlCmd%(Sqlconn%, " AND type = 'S'")
```

## See Also

[SqlFreeBuf](#); [DB-Library for Visual Basic Options](#)

# SqlCmdRow%

Indicates whether the current statement can return rows.

## Syntax

**SqlCmdRow%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlCmdRow%** determines whether the current command can return rows. The following types of statements can return rows:

- A Transact-SQL SELECT statement
- A Transact-SQL EXECUTE statement that is executing a stored procedure containing a SELECT statement

Call **SqlCmdRow%** after **SqlResults%** returns SUCCEED. Even if **SqlCmdRow%** returns SUCCEED, the statement will not return rows if none have qualified. To determine whether any rows are returned, use **SqlRows%**.

Even if **SqlCmdRow%** returns FAIL, you must still process the results by calling **SqlNextRow%** until it returns NOMOREROWS.

## See Also

[SqlNextRow%](#), [SqlResults%](#), [SqlRows%](#), [SqlRowType%](#)

# SqlCollInfo%

Returns information about a regular column or a compute column in a result set, or a column in a cursor.

## Syntax

**SqlCollInfo%** ( *sqlhandle%*, *type%*, *column%*, *computeid%*, *columninfo* )

where

### *sqlhandle%*

Is a SQL Server connection or a cursor handle. If *type%* is **SQLCI\_REGULAR%** or **SQLCI\_ALTERNATE%**, this is a SQL Server connection (returned by **SqlOpen%**). If *type%* is **SQLCI\_CURSOR%**, this is a cursor handle (returned by **SqlCursorOpen%**).

### *type%*

Is the type of column information to return. The following table describes the different *type%* values:

<b>type%</b>	<b>Description</b>
<b>SQLCI_REGULAR%</b>	Return information about a regular column in the current result set.
<b>SQLCI_ALTERNATE%</b>	Returns information about a compute column in the current result set.
<b>SQLCI_CURSOR%</b>	Returns information about a column in the open cursor.

### *column%*

The number of the column. The first column is number 1.

### *computeid%*

If *type%* is **SQLCI\_ALTERNATE**, this is the ID that identifies the compute (the result of a COMPUTE clause) value. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

### *columninfo*

A *ColInfo* structure that DB-Library will use to return detailed information about the specified column.

The *ColInfo* structure is defined as follows:

```
Type ColInfo
    Name As String * 31
    ActualName As String * 31
    TableName As String * 31
    CType As Integer
    UserType As Long
    MaxLength As Long
    Precision As Integer
    Scale As Integer
    VarLength As Integer
    Null As Integer
    CaseSensitive As Integer
    Updatable As Integer
    Identity As Integer
End Type
```

The *ColInfo* fields (and the equivalent DB-Library functions, if any) are described below:

<b>Field</b>	<b>Description</b>
<b>Name</b>	Is the name of the returned column ( <b>SqlColName\$</b> ).

<b>ActualName</b>	Is the actual name of the column in <b>TableName</b> if <i>type%</i> is SQLCI_CURSOR%; otherwise, this is the same as <b>Name</b> .
<b>TableName</b>	Is the table that contains the column if <i>type%</i> is SQLCI_CURSOR%; otherwise, an empty string.
<b>CType</b>	Is the datatype of the column ( <b>SqlColType%</b> , <b>SqlAltType%</b> ).
<b>UserType</b>	Is the user-defined datatype of the column ( <b>SqlColUType&amp;</b> , <b>SqlAltUType&amp;</b> ).
<b>MaxLength</b>	Is the maximum length, in bytes, of the column ( <b>SqlColLen%</b> , <b>SqlAltLen%</b> ).
<b>Precision</b>	The precision if <b>Type</b> is SQLDECIMAL or SQLNUMERIC; otherwise, - 1.
<b>Scale</b>	The scale if <b>Type</b> is SQLDECIMAL or SQLNUMERIC; otherwise, - 1.
<b>VarLength</b>	Is one of the following: SUCCEED (1) if the column is variable-length. FAIL (0) if the column is fixed-length.
<b>Null</b>	Is one of the following: SUCCEED (1) if the column allows nulls. FAIL (0) if the column does not allow nulls. SQLUNKNOWN (2) if nullability is unknown.
<b>CaseSensitive</b>	Is one of the following: SUCCEED (1) if the column is case sensitive. FAIL (0) if the column is case insensitive. SQLUNKNOWN (2) if case sensitivity is unknown.
<b>Updatable</b>	Is one of the following: SUCCEED (1) if the column can be changed. FAIL (0) if the column is read-only and cannot be changed. SQLUNKNOWN (2) if updatability is unknown.
<b>Identity</b>	Is one of the following: SUCCEED (1) if the column is an identity column. FAIL (0) if the column is not an identity column.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

The **SqlColInfo%** function fills the supplied *ColInfo* structure with information about the specified column. Call this function after **SqlResults%** returns SUCCEED, or after **SqlCursorOpen%** returns a cursor handle.

## See Also

[SqlAltLen%](#), [SqlAltType%](#), [SqlAltUType&](#), [SqlColLen%](#), [SqlColName\\$](#), [SqlColType%](#), [SqlColUType&](#)

# SqlColLen%

Returns the maximum length, in bytes, of the data in a column.

## Syntax

**SqlColLen%** ( *sqlconn%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of the column. The first column is number 1.

## Returns

The maximum length of the data in a specified column. If the column number is not within range of the maximum length of the data, **SqlColLen%** return -1.

## Remarks

**SqlColLen%** do not return the actual length of the data in a column, but rather the maximum length that the data can be. For the actual length of data in a column, use **SqlDatLen&**. Call **SqlColLen%** after **SqlResults%** returns SUCCEED.

## Example

The following code fragment uses **SqlColLen%** to return the maximum length of the data in the *name*, *id*, and *type* columns from the *sysobjects* table:

```
'Put the statement into the command buffer.
cmd$ = "SELECT name, id, type FROM sysobjects"
Result% = SqlCmd$(Sqlconn%, cmd$)

'Send the statement to SQL Server and start execution.
Result% = SqlExec$(Sqlconn%)

'Process the statement results.
Result% = SqlResults$(Sqlconn%)

'Print the column lengths.
FOR ColumnNum% = 1 TO 3
    ColumnLen& = SqlColLen$(Sqlconn%, ColumnNum%)
    PRINT "Column"; ColumnNum%; " length is"; ColumnLen&
NEXT ColumnNum%
```

## Output:

```
Column 1 length is 30
Column 2 length is 11
Column 3 length is 2
```

## See Also

[SqlColName\\$](#), [SqlColType%](#), [SqlData\\$](#), [SqlDatLen&](#), [SqlNumCols%](#)

# SqlColName\$

Returns the name of a result column.

## Syntax

**SqlColName\$** ( *sqlconn%*, *column%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of the column. The first column is number 1.

## Returns

A string containing the name of a result column. If the column number is out of range or if the column is the result of an expression with no name assigned, an empty string is returned.

## Example

The following code fragment uses **SqlColName\$** to return the *name*, *id*, and *type* column names from the *sysobjects* table:

```
'Put the statement into the command buffer.
cmd$ = "SELECT name, id, type FROM sysobjects"
Result% = SqlCmd$(Sqlconn%, cmd$)

'Send the statement to SQL Server and start execution.
Result% = SqlExec$(Sqlconn%)

'Process the statement results.
Result% = SqlResults$(Sqlconn%)

'Print the column names.
FOR ColumnNum% = 1 TO 3
    ColumnName$ = SqlColName$(Sqlconn%, ColumnNum%)
    PRINT "Column"; ColumnNum%; " name is "; ColumnName$
NEXT ColumnNum%
```

## Output:

```
Column 1 name is name
Column 2 name is id
Column 3 name is type
```

## See Also

[SqlColLen%](#), [SqlColType%](#), [SqlData\\$](#), [SqlDatLen&](#), [SqlNumCols%](#)

# SqlColType%

Returns the SQL Server datatype for a result column.

## Syntax

**SqlColType%** ( *sqlconn%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of the column. The first column is number 1.

## Returns

An integer value for the datatype of a specified column. If the column number is not in range, -1 is returned.

These are the returned values:

Column datatype	Returned constant
<i>char</i>	SQLCHAR
<i>varchar</i>	SQLCHAR
<i>binary</i>	SQLBINARY
<i>varbinary</i>	SQLBINARY
<i>tinyint</i>	SQLINT1
<i>smallint</i>	SQLINT2
<i>int</i>	SQLINT4
<i>real</i>	SQLFLT4
<i>float</i>	SQLFLT8
<i>smallmoney</i>	SQLMONEY4
<i>money</i>	SQLMONEY
<i>decimal</i>	SQLDECIMAL
<i>numeric</i>	SQLNUMERIC
<i>smalldatetime</i>	SQLDATETIME4
<i>datetime</i>	SQLDATETIME
<i>image</i>	SQLIMAGE
<i>text</i>	SQLTEXT

## Remarks

The **SqlColType%** function returns an integer value for the type. Use **SqlPrType%** to convert the type value into a readable string. This function cannot determine whether a column allows null values.

## Example

```
'Put the statement into the command buffer.  
cmd$ = "SELECT name, id, type FROM sysobjects"  
Result% = SqlCmd$(Sqlconn%, cmd$)
```



```
'Send the statement to SQL Server and start execution.
Result% = SqlExec%(Sqlconn%)

'Process the statement results.
Result% = SqlResults%(Sqlconn%)

'Print the column type.
FOR ColumnNum% = 1 TO 3
    ColumnType% = SqlColType%(Sqlconn%, ColumnNum%)
    PRINT "Column"; ColumnNum%; " type is"; ColumnType%
NEXT ColumnNum%
```

**See Also**

[SqlColLen%](#), [SqlColName\\$](#), [SqlData\\$](#), [SqlDatLen&](#), [SqlNumCols%](#)

# SqlColUType&

Returns the user-defined datatype of a regular result column.

## Syntax

**SqlColUType&** ( *sqlconn%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of the column. The first column returned is number 1.

## Returns

The user-defined datatype of the specified compute column on success, or - 1 on error.

## Remarks

Call **SqlColUType&** only after **SqlNextRow%** or **SqlGetRow%** returns SUCCEED.

## See Also

[SqlData\\$](#), [SqlGetRow%](#), [SqlNextRow%](#), [SqlNumCols%](#)

# SqlCount&

Returns the number of rows affected by the current statement.

## Syntax

**SqlCount&** ( *sqlconn%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of rows affected by the current statement. Call **SqlIsCount%** to determine if this count is correct.

## Remarks

After the results of a Transact-SQL statement are processed, use **SqlCount&** to find out how many rows have been affected by the statement. For example, if a SELECT statement is sent to SQL Server and you read all the rows by calling **SqlNextRow%** until it returns NOMOREROWS, you can call **SqlCount&** to find out how many rows have been retrieved. If the current Transact-SQL statement does not return rows (the DELETE statement, for example), call **SqlCount&** after **SqlResults%**.

If the current statement executes a stored procedure, for example an EXEC or a call to a remote stored procedure, **SqlCount&** reports the number of rows returned by the last SELECT statement executed by the stored procedure. Note that a stored procedure without a SELECT statement can execute a SELECT anyway simply by calling another stored procedure with a SELECT.

## See Also

[SqlIsCount%](#), [SqlNextRow%](#), [SqlResults%](#)

# SqlCurCmd%

Returns the number of the current statement in the command buffer.

## Syntax

**SqlCurCmd%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of the current statement. The first statement in a group of statements is number 1.

## Remarks

The statement number is incremented every time **SqlResults%** returns SUCCEED or FAIL. Unsuccessful statements are counted. The statement number is reset by each call to **SqlExec%** or **SqlSend%**.

## See Also

[SqlCmdRow%](#), [SqlExec%](#), [SqlMoreCmds%](#), [SqlResults%](#), [SqlRows%](#), [SqlSend%](#)

# SqlCurRow&

Returns the number of the row currently being read.

## Syntax

**SqlCurRow&** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of the current row.

## Remarks

**SqlCurRow&** returns the number of the row most recently read by **SqlGetRow%** or **SqlNextRow%**. When you do not turn on row buffering, **SqlFirstRow&**, **SqlCurRow&**, and **SqlLastRow&** always return the same value (the number of the current row).

Use **SqlGetRow%** to read rows from the row buffer. The first row SQL Server returns from the row buffer is number 1. The row number changes every time **SqlNextRow%** or **SqlGetRow%** returns SUCCEED. The row number is reset to 0 by each call to **SqlResults%**.

## See Also

[SqlClrBuf](#), [SqlFirstRow&](#), [SqlGetRow%](#), [SqlLastRow&](#), [SqlNextRow%](#), [SqlSetOpt%](#); [DB-Library for Visual Basic Options](#)

# SqlData\$

Returns a string containing data in a result column.

## Syntax

**SqlData\$** ( *sqlconn%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of the result column. The first column is number 1.

## Returns

A string containing the data in a result column. For the SQL Server datatypes *binary*, *varbinary*, and *image*, **SqlData\$** returns a string of binary data, with one character in the string per byte of data in the result column. For all other datatypes, **SqlData\$** returns a string of readable characters. When there is no such column or when the data is NULL, an empty string is returned. To make sure that the data really is NULL, always check for a return of 0 by using **SqlDatLen&**.

## Remarks

Use **SqlDatLen&** to get the length of the data for variable-length datatypes. For numeric datatypes, use the Visual Basic **LEN** function to determine the length of the string returned by **SqlData\$**.

To convert the data from a string to a different datatype, use **SqlColType%** to get the datatype of the data, then use Visual Basic functions to convert to Visual Basic datatypes. Some helpful Visual Basic conversion functions are shown in the following table. For more information, see your documentation for Visual Basic.

Visual Basic function	Description
CDBL	Converts a numeric expression to a double-precision value.
CLNG	Converts a numeric expression to a long integer.
VAL	Converts a string representation of a value to a numeric expression.

## Example

```
'Put the statement into the command buffer.
Result% = SqlCmd%(Sqlconn%, "SELECT id FROM sysobjects")

'Send the statement to SQL Server and start execution.
Result% = SqlExec%(Sqlconn%)

'Process the statement results.
Result% = SqlResults%(Sqlconn%)

'Retrieve and print the data in each row.
RowNum% = 0
DO UNTIL SqlNextRow%(Sqlconn%) = NOMOREROWS
    RowNum% = RowNum% + 1
```

```
PRINT "Row"; RowNum%;" object id is ";  
PRINT SqlData$(Sqlconn%, 1)  
LOOP
```

**See Also**

[SqlColLen%](#), [SqlColName\\$](#), [SqlColType%](#), [SqlDatLen&](#), [SqlNumCols%](#)

# SqlDataReady%

Indicates whether SQL Server has completed processing a command.

## Syntax

**SqlDataReady%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0). SUCCEED means that the data is available to be read.

## Remarks

With **SqlDataReady%**, you can write an application that can continue processing while SQL Server is actually performing the database operation.

**SqlDataReady%** is ordinarily used after a call to **SqlSend%** and before a call to **SqlOk%**. After **SqlSend%**, SQL Server begins executing the statements in the command buffer. When **SqlOk%** is called, DB-Library for Visual Basic waits for SQL Server to complete processing before returning control to the application.

**SqlDataReady%** provides a way to determine when statement processing is complete. Call **SqlDataReady%** repeatedly until it returns a value other than 0. At that point, you can call **SqlOk%**.

**Important** Unless you provide a way for your application to time out, **SqlDataReady%** can return FAIL indefinitely if another process causes a conflicting lock or if the connection is broken.

## See Also

[SqlOk%](#), [SqlResults%](#), [SqlSend%](#)



# SqlDateCrack%

Converts a string of date and time values into a format more usable to the user.

## Syntax

**SqlDateCrack%** ( *sqlconn%*, *dateinfo*, *datetime\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*dateinfo*

Identifies a structure containing the components of the *datetime\$* string. The *dateinfo* structure contains the following fields:

Field	Description
<i>year%</i>	A number of a year in the range 1753 through 9999.
<i>quarter%</i>	A number of a quarter of a year in the range 1 through 4.
<i>month%</i>	A number of a month in the range 1 through 12.
<i>dayofyear%</i>	A number of a day of a year in the range 1 through 366. Leap years are counted.
<i>day%</i>	A number of a day of a month in the range 1 through 31.
<i>week%</i>	A number of a week of a year in the range 1 through 54. Leap years are counted.
<i>weekday%</i>	A number of the day of a week in the range 1 through 7 (Monday through Sunday).
<i>hour%</i>	A number of an hour in the range 0 through 23.
<i>minute%</i>	A number of a minute in the range 0 through 59.
<i>second%</i>	A number of a second in the range 0 through 59.
<i>millisecond%</i>	A number of a millisecond in the range 0 through 999.

*datetime\$*

Is a string containing the date and time.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlDateCrack%** converts a SQL Server DATETIME string into its integer components and puts them into a *dateinfo* structure.

Date and time values are maintained in an internal format that is not readily usable. For example, a time value is stored as the number of 300ths of a second since midnight, and a date value is stored as the number of days since January 1, 1900. The **SqlDateCrack%** function converts the internal value to something easily usable by an application.

## Example

```
'Put the statement into the command buffer.
```

```
Result% = SqlCmd%(Sqlconn%, "SELECT name, crdate FROM master..sysdatabases")
```

```
'Send the statement to SQL Server and start execution.
```

```

Result% = SqlExec%(Sqlconn%)

'Process the statement results.
Result% = SqlResults%(Sqlconn%)

'Retrieve and print the database name and its date info.
DO UNTIL SqlNextRow%(Sqlconn%) = NOMOREROWS
    PRINT "Database Name is "
    PRINT SqlData$(Sqlconn%, 1)
    PRINT
    PRINT "Creation date string info is "
    PRINT SqlData$(Sqlconn%, 2)
    PRINT

    'Break up the creation date into its constituent parts.

    Datetime$ = SqlData$(Sqlconn%, 2)

    SqlDateCrack(Sqlconn%, Dateinfo(), Datetime$)

    'Print the parts of the creation date.
    PRINT
    PRINT "Year = "; Dateinfo.year
    PRINT "Month = "; Dateinfo.month
    PRINT "Day of month = "; Dateinfo.day
    PRINT "Day of year = "; Dateinfo.dayofyear
    PRINT "Day of week = "; Dateinfo.weekday
    PRINT "Hour = "; Dateinfo.hour
    PRINT "Minute = "; Dateinfo.minute
    PRINT "Second = "; Dateinfo.second
    PRINT "Millisecond = "; Dateinfo.millisecond

LOOP

```

## See Also

[SqlData\\$](#)

# SqlDatLen&

Returns the actual length, in bytes, of the data in a column.

## Syntax

**SqlDatLen&** ( *sqlconn%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of a column. The first column is number 1.

## Returns

The actual length of the data in a specified column. If the column has a null value, **SqlDatLen&** returns 0. If the column number is out of range, **SqlDatLen&** returns -1. Calling **SqlDatLen&** after **SqlNextRow%** or **SqlGetRow%** returns Regrow.

## Remarks

For numeric datatypes such as *smallint* and *float*, **SqlDatLen&** returns the maximum printable width, not the length of the string that would be returned by **SqlData\$**. For example, if you store the value 10,000 in a *smallint* column, **SqlDatLen&** returns 6 (the storage size of the *smallint* datatype). The length of the string returned by **SqlData\$**, in this case 5, is determined by using the Visual Basic **LEN** function.

You obtain the maximum possible length for the data in a column by calling **SqlColLen%**. The data itself is returned by **SqlData\$**.

## Example

```
'Put the statement into the command buffer.
Result% = SqlCmd%(Sqlconn%, "SELECT name FROM sysobjects")

'Send the statement to SQL Server and start execution.
Result% = SqlExec%(Sqlconn%)

'Process the statement results.
Result% = SqlResults%(Sqlconn%)

'Print the data length of each row.
RowNum% = 0
DO UNTIL SqlNextRow%(Sqlconn%) = NOMOREROWS
    RowNum% = RowNum% + 1
    PRINT "Row"; RowNum%; " data length is ";
    PRINT SqlDatLen&(Sqlconn%, 1)
LOOP
```

## Output:

```
Row 1 data length is 10
Row 2 data length is 10
.
.
.
```

Row 103 data length is 13

**See Also**

[SqlColLen%](#), [SqlColName\\$](#), [SqlColType%](#), [SqlData\\$](#), [SqlNumCols%](#)

# SqlDead%

Indicates whether a SQL Server connection is inactive.

## Syntax

**SqlDead%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlDead%** is particularly useful in user-supplied error handlers. If the SQL Server connection is dead, almost every DB-Library for Visual Basic procedure that receives that connection as a parameter immediately fails, calling the error handler. You must close an inactive *sqlconn%* connection with **SqlClose%** and open a new connection with **SqlOpen%**.

**SqlDead%** usually returns true when a network connection is broken. Note any other errors or messages that you receive.

## See Also

[Programming with DB-Library for Visual Basic; Error Messages](#)

# SqlExit

Closes and frees all SQL Server connections.

## Syntax

**SqlExit**

## Remarks

**SqlExit** calls **SqlClose** repeatedly for all allocated SQL Server connections. **SqlClose** stops any activity associated with a SQL Server connection, closes a SQL Server connection, and frees allocated memory. SQL Server connections are opened with **SqlOpen%**.

You can use **SqlClose** directly to close a single SQL Server connection.

## See Also

[SqlClose](#), [SqlOpen%](#)

# SqlFirstRow&

Returns the number of the first row in the row buffer.

## Syntax

**SqlFirstRow&** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of the first row in the row buffer. The first row returned is number 1. Use this return value when making a call to **SqlGetRow%**.

## Remarks

If you do not turn on row buffering, **SqlFirstRow&**, **SqlCurRow&**, and **SqlLastRow&** always return the same value (the number of the current row).

If you turn on row buffering by setting the SQLBUFFER option with **SqlSetOpt%**, **SqlFirstRow&** returns the number of the lowest (oldest) row of results in the buffer. For example, if a query returns 100 result rows and you read 20 rows into the buffer using **SqlNextRow%**, **SqlFirstRow&** returns 1, regardless of which row is current. As the application processes and clears rows from the buffer using **SqlClrBuf** and more data is read from the server using **SqlNextRow%**, **SqlFirstRow&** returns the number of the result row stored in the lowest (oldest) buffer location.

**SqlLastRow&** returns the number of the result row stored in the highest (newest) buffer location.

## See Also

[SqlClrBuf](#), [SqlCurRow&](#), [SqlGetRow%](#), [SqlLastRow&](#), [SqlNextRow%](#), [SqlSetOpt%](#); [DB-Library for Visual Basic Options](#)

# SqlFreeBuf

Clears the command buffer.

## Syntax

**SqlFreeBuf** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Remarks

This function frees any space allocated to the command buffer of the *sqlconn%* structure. Statements for SQL Server are added to the command buffer with **SqlCmd%**. After a call to **SqlExec%** or **SqlSend%**, the first call to **SqlCmd%** automatically calls **SqlFreeBuf** to clear the command buffer before the new text is entered. If you don't want the buffer automatically cleared, set the SQLNOAUTOFREE option using **SqlSetOpt%**. When SQLNOAUTOFREE is set, the command buffer is cleared only by a call to **SqlFreeBuf**.

You can access the contents of the command buffer by using **SqlGetChar\$**, **SqlStrLen%** and **SqlStrCpy%**.

## See Also

[SqlCmd%](#), [SqlExec%](#), [SqlSend%](#), [SqlStrCpy%](#), [SqlStrLen%](#); [DB-Library for Visual Basic Options](#)



# SqlFreeLogin

Frees the memory allocated by **SqlLogin%** for a login record.

## Syntax

**SqlFreeLogin** ( *loginrec%* )

where

*loginrec%*

Is a login record. The value of *loginrec%* is returned by **SqlLogin%**.

## Remarks

You can call **SqlFreeLogin** immediately after you call **SqlOpen**, and you can use the same login record for multiple calls to **SqlOpen**. Call **SqlFreeLogin** when you are completely finished with a login record.

## Example

```
Login% = SqlLogin%  
Result% = SqlSetLUser%(Login%, "loginid")  
Result% = SqlSetLPwd%(Login%, "passwd")  
Result% = SqlSetLApp%(Login%, "myapp")
```

```
Sqlconn% = SqlOpen%(Login%, "server")
```

```
CALL SqlFreeLogin(Login%)
```

## See Also

[SqlLogin%](#), [SqlOpen%](#)

# SqlGetChar\$

Returns the value of a character in the command buffer.

## Syntax

**SqlGetChar\$** ( *sqlconn%*, *charnum%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*charnum%*

Is the character to find in the command buffer. The first character is the 0th character.

## Returns

The value of *charnum%* in the command buffer. If *charnum%* is not in range, an empty string is returned.

## Remarks

Use **SqlGetChar\$** to find a particular character in the command buffer. It returns a string of the *charnum%* character in the command buffer.

Internally, the command buffer is a linked list of non - null-terminated text strings. Parts of the command buffer can be located and copied using the **SqlGetChar\$**, **SqlStrCpy%**, and **SqlStrLen%** functions.

## See Also

[SqlCmd%](#), [SqlFreeBuf](#), [SqlStrCpy%](#), [SqlStrLen%](#)

# SqlGetMaxProcs%

Determines the current maximum number of simultaneously open *sqlconn* connections.

## Syntax

**SqlGetMaxProcs%** ( )

## Returns

An integer representing the current limit on the number of simultaneously open *sqlconn* connections.

## Remarks

A DB-Library for Visual Basic program has a maximum number of simultaneously open *sqlconn* connections. The initial default limit is set to 25. The application program can change this limit by calling **SqlSetMaxProcs%**.

## See Also

[SqlOpen%](#), [SqlSetMaxProcs%](#)

# SqlGetOff%

Checks for the existence of Transact-SQL statements in the command buffer.

## Syntax

**SqlGetOff%** ( *sqlconn%*, *offtype%*, *startfrom%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*offtype%*

Is the type of offset you want to find. The types are OFF\_SELECT, OFF\_FROM, OFF\_ORDER, OFF\_COMPUTE, OFF\_TABLE, OFF\_PROCEDURE, OFF\_STATEMENT, OFF\_PARAM, and OFF\_EXEC.

For details, see [DB-Library for Visual Basic Options](#).

*startfrom%*

Is the point in the buffer from which to start looking. The command buffer begins at 0.

## Returns

The character offset into the command buffer for a specified offset. If the offset is not found, -1 is returned.

## Remarks

If the SQLOFFSET option has been set (see [DB-Library for Visual Basic Options](#).), **SqlGetOff%** can check for the location of certain Transact-SQL statements in the command buffer.

## Examples

In example A, assume that the program doesn't know the contents of the command buffer but needs to know where the SQL keyword SELECT appears:

```
A.  Dim SelectOffset (9) As Integer
    Dim LastOffset As Integer
    Dim i AS Integer
    'Set the offset option.
    SqlSetOption% (Sqlconn%, SQLOFFSET, "select")

    'Execute the option on the server.
    SqlExec%(Sqlconn%)

    'Read the returned results.
    Do Until SqlResults%(Sqlconn%) = NOMORERESULTS

        'Assume the command buffer contains the following SELECTs:
        Result% = SqlCmd%(Sqlconn%, "SELECT x = 100 SELECT y = 5")

    'Send the statement to SQL Server and start execution.
    Result% = SqlExec%(Sqlconn%)

    'Process the statement results.
    Result% = SqlResults%(Sqlconn%)

    'Get all the offsets to the SELECT keyword.
```

```

        LastOffset% = 0
        i% = 0
        Do Until LastOffset% = -1
            LastOffset% = SqlGetOff(Sqlconn%, OFF_SELECT, LastOffset%)
            SelectOffset% (i) = LastOffset% + 1
            i% = i% + 1
        Loop
    Loop

```

In example B, the function **SqlGetOff%** does not recognize SELECT statements in a subquery. So, if the command buffer contains the following program fragment, the second SELECT statement goes unrecognized:

```

B.    select pub_name
        from publishers
        where pub_id not in
        (select pub_id
        from titles
        where type = "business")

```

#### See Also

[SqlCmd%](#), [SqlGetChar\\$](#), [SqlSetOpt%](#), [SqlStrCpy%](#), [SqlStrLen%](#); [Text and image Functions](#)

# SqlGetPacket%

Returns the tabular data stream (TDS) packet size currently in use.

## Syntax

**SqlGetPacket%** ( *sqlconn%* )

where

*sqlconn%*

A SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The TDS packet size currently in use.

## Remarks

To determine the TDS packet size in use, an application should call **SqlGetPacket%** after **SqlOpen%**.

You can change the TDS packet size by using **SqlSetLPacket%**, which sets the packet size field in the login record.

If the call to **SqlGetPacket%** fails (for example, when *sqlconn%* is null), **SqlGetPacket%** returns 0. Other than the failure case of 0, however, the call should never return less than 512 bytes.

## See Also

[SqlSetLPacket%](#)

# SqlGetRow%

Sets the current row in the row buffer to a specific row number and reads it.

## Syntax

**SqlGetRow%** ( *sqlconn%*, *row&* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*row&*

Is the number of the row to read. The first row returned is number 1.

## Returns

One of four types of values depending on certain conditions:

- If the current row is a regular row, REGROW (-1) is returned.
- If the current row is a compute row, the ID of the COMPUTE clause is returned.
- If the function is unsuccessful, FAIL (0) is returned.
- If the row is not in the row buffer, NOMOREROWS (-2) is returned.

## Remarks

**SqlGetRow%** works only if the SQLBUFFER option is set using **SqlSetOpt%**. When buffering is turned off, each row is processed by repeatedly calling **SqlNextRow%** until it returns NOMOREROWS. When buffering is turned on, you can use **SqlGetRow%** to jump to any row that has already been read and is still in the row buffer. After you call **SqlGetRow%** to read a row, you can call **SqlNextRow%** to return rows in order following the row read by **SqlGetRow%**.

## See Also

[SqlClrBuf](#), [SqlNextRow%](#), [SqlSetOpt%](#); [DB-Library for Visual Basic Options](#)

# SqlGetTime%

Returns the number of seconds that DB-Library for Visual Basic waits for a SQL Server response to a Transact-SQL statement.

## Syntax

**SqlGetTime%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of seconds that DB-Library for Visual Basic waits for a SQL Server response to a Transact-SQL statement. A value of 0 (the default) represents an infinite timeout period.

## Remarks

**SqlGetTime%** returns the number of seconds that DB-Library for Visual Basic waits for a SQL Server response to calls to **SqlExec%**, **SqlOk%**, **SqlResults%**, and **SqlNextRow%**. You can change the timeout value by calling **SqlSetTime%**.

## See Also

[SqlSetTime%](#), [SqlExec%](#), [SqlOk%](#), [SqlResults%](#), [SqlNextRow%](#)



# Sqllnit\$

Initializes DB-Library for Visual Basic.

## Syntax

**Sqllnit\$ ( )**

## Returns

A string containing the version number of DB-Library for Visual Basic. If initialization is unsuccessful, **Sqllnit\$** returns an empty string.

**Important** If **Sqllnit\$** returns an empty string, do not attempt to call any other DB-Library for Visual Basic procedures because unpredictable results can occur.

## Remarks

You must call **Sqllnit\$** before calling any other DB-Library for Visual Basic procedures.

**Sqllnit\$** initializes the user-defined error and message handlers, if present. When running with the Windows environment, DB-Library maintains information about each application that has referenced it. DB-Library creates the information when a library application calls **Sqllnit\$**; it does this to prevent conflicts between applications that use DB-Library concurrently. In order for DB-Library to release this information, the application must call **SqlWinExit** just before it exits.

## See also

[SqlOpen%](#) and [SqlWinExit](#)

# SqllsAvail%

Determines whether a *sqlconn* connection is available for general use.

## Syntax

**SqllsAvail%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCESS (1) if the *sqlconn* connection is available for general use; otherwise, FAIL (0).

## Remarks

**SqllsAvail%** indicates whether the specified *sqlconn* connection is available for general use. When a *sqlconn* connection is first opened, it is marked as being available until it is used. Many DB-Library for Visual Basic functions automatically set *sqlconn* to "not available," but only **SqlSetAvail** resets it to "available." This facility is useful when several parts of an application are trying to share a single *sqlconn* connection.

## See Also

[SqlSetAvail](#)

# SqllsCount%

Indicates whether or not the count returned by **SqlCount&** is real.

## Syntax

**SqllsCount%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCESS (1) if the count returned by **SqlCount&** is real or FAIL (0) if the count returned by **SqlCount&** is not real.

## Remarks

You can have commands that return or affect 0 or more rows and commands that do not affect rows. In both cases **SqlCount&** returns 0.

Calling **SqllsCount%** after **SqlCount&** determines whether the count is real.

## See Also

[SqlCount&](#)

# SqllsOpt%

Checks the status of an option set by **SqlSetOpt%**.

## Syntax

**SqllsOpt%** ( *sqlconn%*, *opt%*, *optparam\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**. Unlike the value of *sqlconn%* when you use **SqlSetOpt%** and **SqlClrOpt%**, the value of *sqlconn%* cannot be 0 when you use **SqllsOpt%**.

*opt%*

Is the option to be checked. Do not enclose the option name in quotation marks. For a list of the options available in DB-Library for Visual Basic, see [DB-Library for Visual Basic Options](#).

*optparam\$*

Is a parameter for an option. You must include *optparam\$* for all options, whether or not they take parameters. Enclose *optparam\$* in quotation marks.

If an option does not take a parameter, *optparam\$* is ignored. If an option does take a parameter, *optparam\$* is ignored for all options except SQLOFFSET and SQLSTAT.

The SQLOFFSET and SQLSTAT options can have several settings, each with a different parameter. In these cases, **SqllsOpt%** needs a valid *optparam\$* to determine which option parameter to check.

For a list of the options available in DB-Library for Visual Basic, see [DB-Library for Visual Basic Options](#).

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

Although you can set and clear SQL Server query options directly through Transact-SQL, design your application to use **SqlSetOpt%** and **SqlClrOpt%** to set and clear options because these functions provide a uniform interface for setting both SQL Server and Visual Basic options. With these functions, your application can also use **SqllsOpt%** to check the status of an option. For a list of each option in DB-Library for Visual Basic and its default status, see [DB-Library for Visual Basic Options](#).

## See Also

[SqlClrOpt%](#), [SqlSetOpt%](#); [DB-Library for Visual Basic Options](#)

# SqlLastRow&

Returns the number of the last row in the row buffer.

## Syntax

**SqlLastRow&** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of the last row in the row buffer. The first row returned from SQL Server is number 1.

## Remarks

If you do not turn on row buffering, **SqlFirstRow&**, **SqlCurRow&**, and **SqlLastRow&** always return the same value (the number of the current row).

If you turn on row buffering by setting the SQLBUFFER option with **SqlSetOpt%**, **SqlLastRow&** returns the number of the highest (newest) row of results in the buffer. For example, if a query returns 100 result rows and you read 20 rows into the buffer using **SqlNextRow%**, **SqlLastRow&** returns 20 regardless of which row is current. As the application processes and clears rows from the buffer using **SqlClrBuf** and more data is read from the server using **SqlNextRow%**, **SqlLastRow&** returns the number of the result row stored in the highest (newest) buffer location.

**SqlFirstRow&** returns the number of the result row stored in the lowest (oldest) buffer location.

## See Also

[SqlClrBuf](#), [SqlCurRow&](#), [SqlFirstRow&](#), [SqlGetRow%](#), [SqlNextRow%](#), [SqlSetOpt%](#); [DB-Library for Visual Basic Options](#)

# SqlLogin%

Allocates a login record for use with **SqlOpen%**.

## Syntax

**SqlLogin% ( )**

## Returns

The identifier of a login record. If the login record cannot be allocated, 0 is returned.

**Important** Do not modify this identifier in any way. Modifying the identifier can cause unpredictable results.

## Remarks

Your program can use the following functions to supply the components of a login record:

- **SqlSetLUser%** supplies the login ID.
- **SqlSetLPwd%** supplies the user's password. The password is required only if the user has a password on SQL Server.
- **SqlSetLHost%** supplies the workstation name.
- **SqlSetLApp%** supplies the application name.
- **SqlSetLNatLang%** supplies the national language.
- **SqlSetLPacket%** supplies the TDS packet size.
- **SqlSetLSecure%** requests a secure connection.
- **SqlSetLVersion%** sets DB-Library 6.0 behavior.
- **SqlBCPSetL%** enables bulk copy operations.

## See Also

[SqlBCPSetL%](#), [SqlFreeLogin](#), [SqlOpen%](#), [SqlSetLApp%](#), [SqlSetLHost%](#), [SqlSetLPacket%](#), [SqlSetLPwd%](#), [SqlSetLSecure%](#), [SqlSetLUser%](#), [SqlSetLVersion%](#)

# SqlMoreCmds%

Indicates whether there are more statements in the command buffer to be processed.

## Syntax

**SqlMoreCmds%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0). SUCCEED indicates there are more results in the command buffer to be processed.

## Remarks

You can call **SqlMoreCmds%** after **SqlNextRow%** returns NOMOREROWS. If you know that the current statement is returning no rows, you can call **SqlMoreCmds%** after **SqlResults%** returns SUCCEED.

You can get the same information by calling **SqlResults%** until it returns NOMORERESULTS.

## See Also

[SqlCmdRow%](#), [SqlResults%](#), [SqlRows%](#), [SqlRowType%](#)

# SqlName\$

Returns the name of the current database.

## Syntax

**SqlName\$** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The name of the current database.

## Remarks

If you need to keep track of when the database changes, use **SqlChange\$**.

## See Also

[SqlChange\\$](#), [SqlUse%](#)



# SqlNextRow%

Reads in the next data row from the row buffer.

## Syntax

**SqlNextRow%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

One of five types of values depending on certain conditions:

- If **SqlNextRow%** reads a regular row, REGROW (-1) is returned. Regular rows contain data from columns designated by a SELECT statement.
- If **SqlNextRow%** is unsuccessful, FAIL (0) is returned.
- If there are no more rows to be read, if the statement returns no rows, or if the server was unable to return more rows (for example, when a deadlock occurs), NOMOREROWS (-2) is returned.
- If row buffering is turned on (using **SqlSetOpt%** with the SQLBUFFER option) and reading the next row would fill the buffer, BUFFULL (-3) is returned. In this case, no row is read, and to continue, at least one row must be cleared from the top of the row buffer. Clear the row buffer by calling **SqlClrBuf**.
- If the current row is a compute row, the identification number of the COMPUTE clause is returned.

## Remarks

Normally, each row is processed in turn by repeatedly calling **SqlNextRow%**. If row buffering is turned on and the row buffer is cleared by **SqlClrBuf**, the discarded rows are no longer available. When row buffering is turned off, the last row is cleared when **SqlNextRow%** returns NOMOREROWS. If row buffering is turned on and you call **SqlGetRow%** to read a row, you can call **SqlNextRow%** to return rows in order following the row read by **SqlGetRow%**.

**SqlResults%** must be called and must return SUCCEED before you make any calls to **SqlNextRow%**.

You must continue calling **SqlNextRow%** until it returns NOMOREROWS. This is true even when you know that your query generates only one results row. Even if **SqlRows%** or **SqlCmdRow%** returns FAIL (indicating that no rows were returned), you must process the results by calling **SqlNextRow%** until it returns NOMOREROWS.

SQL Server can return two types of rows:

- Regular rows containing data from columns designated by a SELECT statement's select list.
- Compute rows resulting from the COMPUTE clause.

To help process data rows from SQL Server, **SqlNextRow%** returns different values according to the type of row. For details, see the previous section, "Returns."

**Note** This function is one of the four that do not return control to the application until the server sends the required response. The application can be blocked for a considerable time if the server is waiting for a lock or is processing a large sort. If this is unacceptable, always call **SqlDataReady%** before **SqlNextRow%** and set the DB-Library timeout to regain control periodically.

## Example

The following program fragment uses **SqlNextRow%** to process a regular row:

```
'Put the statement into the command buffer.
Result% = SqlCmd%(Sqlconn%, "SELECT data FROM table")

'Send the statement to SQL Server and start execution.
Result% = SqlExec%(Sqlconn%)
'Process the statement results.
Result% = SqlResults%(Sqlconn%)

'Retrieve and process the data in each row.
DO UNTIL SqlNextRow%(Sqlconn%) = NOMOREROWS

    'Code to print or process row of data.

LOOP
```

### See Also

[SqlCancel%](#), [SqlCanQuery%](#), [SqlClrBuf](#), [SqlGetRow%](#), [SqlResults%](#), [SqlRows%](#); [DB-Library for Visual Basic Options](#)

# SqlNumAlts%

Returns the number of columns in a COMPUTE clause row.

## Syntax

**SqlNumAlts%** ( *sqlconn%*, *computeid%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*computeid%*

Is the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, each of which can have a different number of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

## Returns

The number of columns for a particular *computeid%*. If *computeid%* is invalid, -1 is returned.

## Remarks

Call this function after **SqlResults%** returns SUCCEED.

## Example

In the following example, **SqlNumAlts%(sqlconn%, 1)** returns 3:

```
select dept, year, sales from employee
order by dept, year
compute avg(sales), min(sales), max(sales) by dept
```

## See Also

[SqlIData\\$, SqlADLen&, SqlAltLen%, SqlAltType%, SqlGetRow%, SqlNextRow%, SqlResults%](#)

# SqlNumCols%

Returns the number of columns in the current set of results.

## Syntax

**SqlNumCols%** ( *sqlconn%* ) ( )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of columns in the current results set. If there are no columns, 0 is returned.

## Example

```
'Put two statements into the command buffer.
cmd$ = "SELECT name, type, id FROM sysobjects"
cmd$ = cmd$ + " SELECT type FROM sysobjects"
Result% = SqlCmd%(Sqlconn%, cmd$)

'Send the statements to SQL Server and start execution.
Result% = SqlExec%(Sqlconn%)

'Process the results of each statement.
DO UNTIL SqlResults%(Sqlconn%) = NOMORERESULTS
    PRINT SqlNumCols%(Sqlconn%);
    PRINT "column(s) in this result."
    DO UNTIL SqlNextRow%(Sqlconn%) = NOMOREROWS
        'Code to print or process row of data.
    LOOP
LOOP
```

## Output:

```
3 column(s) in this result.
1 column(s) in this result.
```

## See Also

[SqlColLen%](#), [SqlColName\\$](#)

# SqlNumCompute%

Returns the number of COMPUTE clauses in the current set of results.

## Syntax

**SqlNumCompute%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of COMPUTE clauses in the current results set.

## Remarks

Call **SqlNumCompute%** after **SqlResults%** returns SUCCEED.

## Example

In this example, **SqlNumCompute%(sqlconn%)** returns 2 because there are two COMPUTE clauses in the SELECT statement:

```
SELECT dept, name FROM employee
ORDER BY dept, name
COMPUTE COUNT(name) BY dept
COMPUTE COUNT(name)
```

## See Also

[SqlNumAlts%](#), [SqlResults%](#)

# SqlNumOrders%

Returns the number of columns specified in the ORDER BY clause of a Transact-SQL SELECT statement.

## Syntax

**SqlNumOrders%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of ORDER BY columns.

## Remarks

Once a SELECT statement is executed and **SqlResults%** returns SUCCEED, call **SqlNumOrders%** to find out how many columns are specified in the current statement's ORDER BY clause.

## See Also

[SqlOrderCol%](#), [SqlResults%](#)

# SqlOpen%

Allocate and initialize a SQL Server connection.

## Syntax

**SqlOpen%** ( *loginrec%*, *server\$* )

where

*loginrec%*

Is a login record. The value of *loginrec%* is returned by **SqlLogin%**.

*server\$*

Is the name of the SQL Server you want to connect to.

## Returns

The identifier of a SQL Server connection. **SqlOpen%** returns 0 if a SQL Server connection cannot be created or initialized, or if your login to SQL Server fails. When 0 is returned, the DB-Library error handler is called to indicate the error.

**Important** Do not modify the identifier returned by **SqlOpen%** in any way. Modifying the **SqlOpen%** identifier can cause unpredictable results.

## Remarks

To communicate with SQL Server, DB-Library for Visual Basic requires a SQL Server connection.

**SqlOpen%** allocates the data structure associated with the connection, sets up communication with the network, logs in to SQL Server, and initializes any default options. **SqlOpen%** returns an identifier that is subsequently used by almost every DB-Library for Visual Basic procedure. A program can open multiple connections with SQL Server. The same login record can be used for multiple calls to **SqlOpen%**.

**SqlOpen%** returns 0 when it encounters any of the errors in the following table:

Error	Description
SQLCONN	Server is unavailable or does not exist.
SQLPWD	Login is incorrect.
SQLSQLPS	Maximum number of SQL Server connections already allocated.

## Example

```
'Get a login record and set login attributes.  
Login% = SqlLogin%()  
Result% = SqlSetLUser%(Login%, "loginid")  
Result% = SqlSetLPwd%(Login%, "passwd")  
Result% = SqlSetLApp%(Login%, "myapp")
```

```
'Get a connection for communicating with SQL Server.
```

```
Sqlconn% = SqlOpen%(Login%, "server")
```

## See Also

[SqlClose](#), [SqlExit](#), [SqlLogin%](#), [SqlSetLoginTime%](#), [Messages](#)





# SqlOrderCol%

Returns the number of the column that appears in a specified location within the ORDER BY clause of the most recently executed Transact-SQL SELECT statement.

## Syntax

**SqlOrderCol%** ( *sqlconn%*, *order%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*order%*

Is the number that identifies a particular ORDER BY column. The first column named in the ORDER BY clause is number 1.

## Returns

The column number (based on the column's position in the select list) for the column in a specified place in the ORDER BY clause. If *order%* is invalid, -1 is returned.

## Remarks

Call **SqlOrderCol%** after **SqlResults%** has returned SUCCEED.

## Example

In the following Transact-SQL statement, **SqlOrderCol%** with an *order%* parameter of 1 returns 3, since the first column named in the ORDER BY clause refers to the third column in the SELECT clause:

```
SELECT dept, name, salary FROM employee  
ORDER BY salary, name
```

## See Also

[SqlNumOrders%](#)

# SqlProcInfo%

Returns information about a SQL Server connection.

## Syntax

**SqlProcInfo%** ( *sqlconn%*, *conninfo* )

where

*sqlconn%*

A SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*conninfo*

A *ProcInfo* structure that DB-Library will fill with information about the specified connection.

The *ProcInfo* structure is defined as follows:

```
Type ProcInfo
    ServerType As Integer
    ServerMajor As Integer
    ServerMinor As Integer
    ServerRevision As Integer
    ServerName As String * 31
    NetLibName As String * 31
    NetLibConnStr As String * 256
End Type
```

The *ProcInfo* fields are described below:

Field	Description
<b>ServerType</b>	Is one of the following: SERVTYPE_MICROSOFT if you are connected to a Microsoft SQL Server. SERVTYPE_UNKNOWN if you are connected to an unknown type of SQL Server.
<b>ServerMajor</b>	Is the XX value in the XX.YY.ZZZ version number of the SQL Server you are connected to. For example, 6.
<b>ServerMinor</b>	Is the YY value in the XX.YY.ZZZ version number of the SQL Server you are connected to. For example, 0.
<b>ServerRevision</b>	Is the ZZZ value in the XX.YY.ZZZ version number of the SQL Server you are connected to. For example, 101.
<b>ServerName</b>	Is the name of the SQL Server you are connected to
<b>NetLibName</b>	Is the name of the Net-Library DLL used to connect to SQL Server.
<b>NetLibConnStr</b>	Is the Net-Library connection string used to connect to SQL Server.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

The **SqlProcInfo%** function fills the supplied *ProcInfo* structure with information about the SQL Server connection.

**See Also**

**[SqlOpen%](#)**

# SqlPrType\$

Converts a SQL Server token value to a string.

## Syntax

**SqlPrType\$** ( *token%* )

where

*token%*

Is the SQL Server token value.

## Returns

A string that is the readable translation of the SQL Server token value. When *token%* is unknown, **SqlPrType\$** returns an empty string. The strings correspond to SQL Server datatype names.

## Remarks

Functions such as **SqlColType%** and **SqlAltType%** return SQL Server token values. Use **SqlPrType\$** to get the string translation of the token value.

The following table shows the token values used by **SqlPrType\$**:

Token value	Datatype
SQLINT1	<i>tinyint</i>
SQLINT2	<i>smallint</i>
SQLINT4	<i>int</i>
SQLMONEY	<i>money</i>
SQLMONEY4	<i>smallmoney</i>
SQLFLT4	<i>real</i>
SQLFLT8	<i>float</i>
SQLDATETIME	<i>datetime</i>
SQLDATETIME4	<i>smalldatetime</i>
SQLBIT	<i>bit</i>
SQLCHAR	<i>char</i>
SQLVARCHAR	<i>varchar</i>
SQLTEXT	<i>text</i>
SQLBINARY	<i>binary</i>
SQLVARBINARY	<i>varbinary</i>
SQLIMAGE	<i>image</i>
SQLDECIMAL	<i>decimal</i>
SQLNUMERIC	<i>numeric</i>
SQLINTN	<i>integer-null</i>
SQLDATETIMN	<i>datetime-null</i>
SQLMONEYN	<i>money-null</i>
SQLFLTN	<i>float-null</i>
SQLAOPSUM	<i>sum</i>
SQLAOPAVG	<i>avg</i>
SQLAOPCNT	<i>count</i>

SQLAOPMIN	<i>min</i>
SQLAOPMAX	<i>max</i>

**See Also**

[SqlAltOp%](#), [SqlAltType%](#), [SqlColType%](#)

# SqlResults%

Sets up the next statement in the command buffer for processing.

## Syntax

**SqlResults%** ( *sqlconn%* ) ( )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1), FAIL (0), NOMORERESULTS (2), or NOMORERPCRESULTS (3). The most common reason for failing is an invalid SQL Server connection. NOMORERPCRESULTS is returned when stored procedure return information is available from one stored procedure in a batch of multiple stored procedures. NOMORERESULTS is returned if there are no more results to be processed.

**Note** This function is one of the four that do not return control to the application until the server sends the required response. The application can be blocked for a considerable time if the server is waiting for a lock or processing a large sort. If this is unacceptable, always call **SqlDataReady%** before **SqlResults%** and set the DB-Library timeout to regain control periodically.

## Remarks

**SqlResults%** is called after **SqlExec%** or **SqlOk%** returns SUCCEED. **SqlResults%** always returns SUCCEED or NOMORERESULTS on the first call if **SqlExec%** or **SqlOk%** has returned SUCCEED, unless a network error or out-of-memory error has occurred. Once **SqlResults%** returns SUCCEED, you typically process any result rows using **SqlNextRow%** unless a network error or out-of-memory error has occurred.

**SqlResults%** must be called for each statement in the command buffer, whether or not the statement returns any rows. If the number of statements in the command buffer is unknown, you can call **SqlResults%** until it returns NOMORERESULTS. You must also call **SqlResults%** once for each stored procedure in the command buffer. However, if the stored procedure contains more than one Transact-SQL SELECT statement, **SqlResults%** must be called once for each statement. The easiest way to do this is to continue to call **SqlResults%** until it returns NOMORERESULTS.

You must call **SqlResults%** until it returns NOMORERESULTS or until any continued use of the connection causes the DB-Library error 10038 "Results Pending."

## Example

```
'Put the statement into the command buffer.
Result% = SqlCmd%(Sqlconn%, "SELECT data FROM table")

'Send the statement to SQL Server and start execution.
Result% = SqlExec%(Sqlconn%)
DO UNTIL Result% = NOMORERESULTS      'Process the statement results.
    Result% = SqlResults%(Sqlconn%)

    'Retrieve and process the data in each row.
    DO UNTIL SqlNextRow%(Sqlconn%) = NOMOREROWS
        'Code to print or process row of data.
    LOOP
LOOP
```

**See Also**

[SqlCancel%](#), [SqlCanQuery%](#), [SqlExec%](#), [SqlNextRow%](#), [SqlOk%](#), [SqlOpen%](#)

# SqlRows%

Indicates whether the current statement returned rows.

## Syntax

**SqlRows%** ( *sqlconn%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

Call **SqlRows%** after **SqlResults%** returns SUCCEED and before you call **SqlNextRow%**; otherwise, **SqlRows%** returns an incorrect value. You can use **SqlCmdRow%** to determine whether the current statement is one that can return rows (that is, whether it is a Transact-SQL SELECT statement or a Transact-SQL EXECUTE statement executing a stored procedure containing a SELECT statement).

Even if **SqlRows%** returns FAIL, you must still process the results by calling **SqlNextRow%** until it returns NOMOREROWS.

## See Also

[SqlCmdRow%](#), [SqlNextRow%](#), [SqlResults%](#), [SqlRowType%](#)



# SqlRowType%

Indicates whether the current result row is a regular row or a COMPUTE row.

## Syntax

**SqlRowType%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

One of four values depending on certain conditions:

- If the current row is a regular row, REGROW (-1) is returned.
- If the current row is a compute row, the ID of the COMPUTE clause is returned.
- If **SqlRowType%** was unsuccessful, FAIL (0) is returned.
- If no rows have been read, NOMOREROWS (-2) is returned.

## Remarks

Because **SqlNextRow%** returns the row type, you can usually determine the type of the current row without using **SqlRowType%**.

## See Also

[SqlNextRow%](#)

# SqlServerEnum%

Lists the names of local SQL Servers, network SQL Servers, or both.

## Syntax

**SqlServerEnum%** ( *searchmode%*, *serverbuf\$*, *numentries%* )

where

### *searchmode%*

Determines whether **SqlServerEnum%** checks for server names locally, on the network, or both.

LOCSEARCH searches for the server names listed in the WIN.INI file. LOCSEARCH has the value 1.

NETSEARCH searches for server names on the network type defined by the default Net-Library. The VBSQL Library loads the default Net-Library set by the DSQUERY entry in the WIN.INI file. If no default entry exists, the VBSQL Library loads the named pipes Net-Library. NETSEARCH has the value 2.

(LOCSEARCH + NETSEARCH) searches for server names using both methods, searching first for server names listed in the WIN.INI file and then for server names on the network type defined by the default Net-Library. A server name is listed twice if it is specified in the WIN.INI file and is also available on the default network type.

### *serverbuf\$*

Is the buffer that stores the server names that **SqlServerEnum%** returns. You must preallocate this buffer using an appropriate Visual Basic function such as **Space\$**. DB-Library for Visual Basic determines the length of the buffer based on what you allocated.

### *numentries%*

Is an output parameter that returns the number of server names copied to the buffer by the current call to **SqlServerEnum%**.

## Returns

**SqlServerEnum%** returns one or more of the following status codes:

Constant	Value
ENUMSUCCESS%	0
MOREDATA%	1
NETNOTAVAIL%	2
OUTOFMEMORY%	4
NOTSUPPORTED%	8

## Remarks

The **SqlServerEnum%** function returns a list of server names to the *server\$* buffer in the order in which it finds them. LOCSEARCH names are listed before NETSEARCH names. Only complete server names are copied to the buffer. Each server name is separated by a null character (**chr\$(0)**).

The end of the list is designated by two consecutive null characters. If the buffer becomes full before **SqlServerEnum%** has finished sending names, **SqlServerEnum%** returns the value MOREDATA.

Note that on named pipes networks, **SqlServerEnum%** returns the server names of computers on which SQL Server is installed, regardless of whether or not SQL Server is currently running.

## See Also

SqlOpen%

# SqlSetAvail

Marks a *sqlconn%* connection as being available for general use.

## Syntax

**SqlSetAvail** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Remarks

Any subsequent call to **SqlIsAvail%** returns SUCCEED until some use is made of the *sqlconn%* connection. **SqlSetAvail** is not normally required in an application.

## See Also

[SqlIsAvail%](#)

# SqlSetLApp%

Sets the application name in the SQL Server login record.

## Syntax

**SqlSetLApp%** ( *loginrec%*, *app\$* )

where

*loginrec%*

Is a login record. The value of *loginrec%* is returned by **SqlLogin%**.

*app\$*

Is the application name sent to SQL Server. This string can have as many as 30 characters. SQL Server stores only the first 16 characters and ignores the rest.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

For **SqlSetLApp%** to have any effect, it must be called before **SqlOpen%**. **SqlSetLApp%** is optional. If **SqlSetLApp%** is not called, the application name is set to an empty string.

SQL Server uses the application name in its *sysprocesses* table to help identify your connection. You see the application name when you query the *sysprocesses* table in the *master* database.

## See Also

[SqlLogin%](#), [SqlOpen%](#), [SqlSetLHost%](#), [SqlSetLPwd%](#), [SqlSetLUser%](#)

# SqlSetLHost%

Sets the workstation name in the SQL Server login record.

## Syntax

**SqlSetLHost%** ( *loginrec%*, *workstation\$* )

where

*loginrec%* ( )

Is a login record. The value of *loginrec%* is returned by **SqlLogin%**.

*workstation\$*

Is the workstation name sent to SQL Server. The string can have as many as 30 characters. SQL Server stores only the first 10 characters and ignores the rest.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

For **SqlSetLHost%** to have any effect, it must be called before **SqlOpen%**. **SqlSetLHost%** is optional. If **SqlSetLHost%** is not called, the workstation name is set to an empty string.

SQL Server uses the workstation name in its *sysprocesses* table and when you issue an **sp\_who** procedure to identify your connection. You see the workstation name when you query the *sysprocesses* table in the *master* database.

## See Also

[SqlLogin%](#), [SqlOpen%](#), [SqlSetLApp%](#), [SqlSetLPwd%](#), [SqlSetLUser%](#)

# SqlSetLNatLang%

Sets the name of the national language in the LOGINREC structure.

## Syntax

**SqlSetLNatLang%** ( *loginrec%*, *language\$* )

where

*loginrec%*

Is a login record to be passed as a parameter to **SqlOpen%**. Execute **SqlLogin%** to get LOGINREC structures.

*language\$*

Is the name of the national language to use.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

To set a language other than the SQL Server default, call **SqlSetLNatLang%** before **SqlOpen%**. If language support is installed on the server, error messages are returned in the designated national language.

## See Also

[SqlLogin%](#) and [SqlOpen%](#)

# SqlSetLoginTime%

Sets the number of seconds that DB-Library for Visual Basic waits for SQL Server to respond to a connection request from **SqlOpen%**.

## Syntax

**SqlSetLoginTime%** ( *timeout%* )

where

*timeout%*

Is the number of seconds that DB-Library for Visual Basic waits for a login response. A *timeout%* value of 0 represents an infinite period. The default *timeout%* value is 60 seconds.

## Returns

SUCCEED (1) or FAIL (0).

## See Also

[SqlSetTime%](#)



# SqlSetLPacket%

Sets the tabular data stream (TDS) packet size in a SQL Server login record.

## Syntax

**SqlSetLPacket%** ( *loginrec%*, *packetsize%* )

where

*loginrec%*

A login record, which is passed as a parameter to **SqlOpen%**. The value of *loginrec%* is returned by **SqlLogin%**.

*packetsize%*

The size requested, in bytes (0 through 65535). The server will set the actual packet size to a value less than or equal to the requested size.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

Tabular data stream (TDS) is an application protocol used for the transfer of requests and request results between clients and servers. TDS data is sent in fixed-size chunks, called "packets." TDS packets have a default size set by SQL Server. If an application does bulk copy operations, or sends or receives large amounts of text or image data, a packet size larger than the default might improve efficiency, since it results in fewer network reads and writes. For large data transfers, a packet size between 4092 and 8192 is usually best. Any larger size can degrade performance.

The only way an application can change the TDS packet size is by using **SqlSetLPacket%**. If **SqlSetLPacket%** is not called, all SQL Server connections in an application will use the default size.

Note that when the application logs in to the server, the server sets the TDS packet size for that SQL Server connection to be equal to or less than the value of the *packetsize%* parameter. If the server is experiencing space constraints, the packet size is set to a value less than the value of the *packetsize%* parameter. Otherwise, the packet size is equal to the value of this parameter. To determine the packet size that the server has set, call **SqlGetPacket%**.

Different SQL Server connections in an application can use different packet sizes. To set different packet sizes for connections, an application can either change the packet size in a single login record between the **SqlOpen%** calls that create the connections, or it can set different packet sizes in multiple login records structures and use these different login records when creating the SQL Server connections.

## See Also

[SqlGetPacket%](#), [SqlLogin%](#), [SqlOpen%](#)

# SqlSetLPwd%

Sets the user's SQL Server password in the SQL Server login record.

## Syntax

**SqlSetLPwd%** ( *loginrec%*, *pwd\$* )

where

*loginrec%* ( )

Is a login record. The value of *loginrec%* is returned by **SqlLogin%**.

*pwd\$*

Is the SQL Server password sent to SQL Server. The string can have as many as 30 characters.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

For **SqlSetLPwd%** to have any effect, it must be called before **SqlOpen%**. **SqlSetLPwd%** is required only if the user has a SQL Server password. If **SqlSetLPwd%** is not called, the password is set to an empty string.

The SQL Server passwords are defined in the *syslogins* table in the *master* database. Only the system administrator has permission to look at passwords.

## See Also

[SqlLogin%](#), [SqlOpen%](#), [SqlSetLApp%](#), [SqlSetLHost%](#), [SqlSetLUser%](#)

# SqlSetLSecure%

Sets the secure connection flag in a SQL Server login record.

## Syntax

**SqlSetLSecure%** ( *loginrec%* )

where

*loginrec%*

A login record, which is passed as a parameter to **SqlOpen%**. The value of *loginrec%* is returned by **SqlLogin%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

By setting the secure connection flag in a login record with **SqlSetLSecure%**, the application requests a secure or *trusted* connection to SQL Server. This means that SQL Server will use integrated login security to establish connections made (using **SqlOpen%**) with this login record, regardless of the current login security mode at the server. Any login ID or password supplied by **SqlSetLUser%** or **SqlSetLPwd%** is ignored.

To use **SqlSetLSecure%** and trusted connections, you must first use **xp\_grantlogin** or SQL Security Manager to grant SQL Server system administrator or user privilege to the appropriate Windows NT - based groups or users. Use **xp\_revokelogin** or SQL Security Manager to revoke SQL Server privileges and stop a user or group from using a trusted connection.

Note that **SqlSetLSecure%** enables trusted connections even when the server is in standard login security mode.

## See Also

[SqlLogin%](#), [SqlOpen%](#), [SqlSetLApp%](#), [SqlSetLHost%](#), [SqlSetLUser%](#)

# SqlSetLUser%

Sets the login ID in the SQL Server login record.

## Syntax

**SqlSetLUser%** ( *loginrec%*, *loginid\$* )

where

*loginrec%* ( )

Is a login record. The value of *loginrec%* is returned by **SqlLogin%**.

*loginid\$*

Is the login ID sent to SQL Server. The string can have as many as 30 characters.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

SQL Server uses the *loginrec%* parameter you set with **SqlSetLUser%** to determine who is attempting a connection. For **SqlSetLUser%** to have any effect, it must be called before **SqlOpen%**. You must provide a login ID to establish a SQL Server connection. The SQL Server login IDs are defined in the *syslogins* table in the *master* database.

## See Also

[SqlLogin%](#), [SqlOpen%](#), [SqlSetLApp%](#), [SqlSetLHost%](#), [SqlSetLPwd%](#)

# SqlSetLVersion%

Sets the DB-Library client behavior to version 4.2 or version 6.0 behavior in a login record.

## Syntax

**SqlSetLVersion%** ( *loginrec%*, *version%* )

where

*loginrec%*

A login record, which is passed as a parameter to **SqlOpen%**. The value of *loginrec%* is returned by **SqlLogin%**.

*version%*

The DB-Library client behavior to use. Must be either SQLVER60% to set DB-Library 6.0 behavior or SQLVER42% to set DB-Library 4.2 behavior.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

If this function is not called, the default is DB-Library version 4.2 behavior.

Using the SQLVER60% value means that SQL Server will treat that connection as a DB-Library 6.0 client in every way. SQL Server will:

- Return *decimal* and *numeric* data values
- Return complete column information (including identity column information) to DB-Library and **SqlCollInfo%**.

Using the SQLVER42% value (or not calling **SqlSetLVersion%** for the login record) means that SQL Server will treat that connection as a DB-Library 4.2x client. SQL Server will:

- Convert *decimal* and *numeric* data values to *float* before returning them to the client
- Return limited, version 4.2 column information (not including identity column information) to DB-Library and **SqlCollInfo%**

Note that using SQLVER60% is not required to use SQL Server 6.0 server cursors.

## See Also

[SqlLogin%](#), [SqlOpen%](#), [SqlSetLApp%](#), [SqlSetLHost%](#), [SqlSetLPwd%](#)

# SqlSetMaxProcs%

Sets the maximum number of simultaneously open *sqlconn* connections.

## Syntax

**SqlSetMaxProcs%** ( *maxprocs%* )

where

*maxprocs%*

Is the new limit on simultaneously open *sqlconn* connections for this particular application.

## Returns

SUCCESS if the call is successful; otherwise, FAIL (*maxprocs%* <= 0).

## Remarks

An application can find out the current connection limit by calling **SqlSetMaxProcs%**. For Windows-based applications, DB-Library for Visual Basic concurrent Windows-based applications can have as many as 45 simultaneously open *sqlconn%* connections. Although the default number is 25, the application can change the default by calling **SqlSetMaxProcs%**.

## See Also

[SqlGetMaxProcs%](#), [SqlOpen%](#)

# SqlSetOpt%

Sets a DB-Library for Visual Basic option.

## Syntax

**SqlSetOpt%** ( *sqlconn%*, *opt%*, *optparam\$* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*opt%*

Is the option to be set. For a list of the options available in DB-Library for Visual Basic, see [DB-Library for Visual Basic Options](#).

*optparam\$*

Is a parameter for an option. Certain options take parameters. For example, the SQLBUFFER option takes as its parameter the number of rows to be buffered. The *optparam\$* variable must always be enclosed in quotation marks, even in the case of a numeric value.

If the option takes no parameters, the value of *optparam\$* is ignored. For options that take no parameters, *optparam\$* can be an empty string or any other string value.

When you use the **Str\$** function to convert a numeric *optparam\$* value to a string, it pads the string with a space. If you don't remove the padded space from the parameter, a call to **SqlSetOpt%** fails. For example, in the following code fragment, **SqlSetOpt%** fails:

```
Optparam$ = Str$(Value%)
SqlSetOpt%(Sqlconn%, X, Optparam$)
```

To use the **Str\$** function to convert a numeric *optparam\$* value to a string, you must use the **LTrim** function to remove the padded space. For example:

```
Optparam$ = Str$(Value%)
SqlSetOpt%(Sqlconn%, X, LTrim$(Optparam$))
```

For a list of the option parameters available in DB-Library for Visual Basic, see [DB-Library for Visual Basic Options](#).

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

Although some options can be set and cleared directly through Transact-SQL, design your application to use **SqlSetOpt%** and **SqlClrOpt%** to set and clear options because **SqlSetOpt%** and **SqlClrOpt%** provide a uniform interface for setting both SQL Server and DB-Library for Visual Basic options. They also allow the application to use **SqlIsOpt%** to check the status of an option.

Three options are unique to DB-Library for Visual Basic: SQLBUFFER, SQLTEXTLIMIT, and SQLNOAUTOFREE. In addition to these options, all Transact-SQL options can be set using **SqlSetOpt%**. If you set any of the Transact-SQL options using **SqlSetOpt%**, DB-Library for Visual Basic places the appropriate Transact-SQL SET statement into the command buffer. This SET statement must be executed at some later time to take effect. Each SET statement means another call to **SqlResults%**. Therefore, design your application to receive the results returned from the statement generated by **SqlSetOpt%**. If an invalid parameter is specified, its invalidity is not known until the statement is sent to SQL Server and the results for that statement are returned using **SqlResults%**.

## See Also

**SqlClrOpt%**, **SqlIsOpt%**, **SqlResults%**; DB-Library for Visual Basic Options



# SqlSetTime%

Sets the number of seconds that DB-Library for Visual Basic waits for a SQL Server response during calls to **SqlExec%**, **SqlOk%**, **SqlResults%**, **SqlNextRow%**, and **SqlRpcExec%**.

## Syntax

**SqlSetTime%** ( *timeout%* )

where

*timeout%*

Is the number of seconds that DB-Library for Visual Basic waits for a SQL Server response before timing out. The default value of 0 represents an infinite timeout period.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlSetTime%** sets the length of time in seconds that DB-Library for Visual Basic waits for a SQL Server response during calls to **SqlExec%**, **SqlOk%**, **SqlResults%**, **SqlNextRow%**, and **SqlRpcExec%**. **SqlSetTime%** can be called at any time before or after a call to **SqlOpen%**. It takes effect immediately upon being called. The **SqlSetTime%** function does not override existing network timeout settings.

To set a timeout value for calls to **SqlOpen%**, use **SqlSetLoginTime%**.

When an application sends a statement to SQL Server using **SqlExec%**, control is not returned to the calling application until SQL Server completes processing the statement or until the timeout expires. If you want your application to continue execution while SQL Server is processing the statement, send the statement with **SqlSend%**, continue its processing, and then, when it is ready to retrieve the results, call **SqlOk%**.

The application can call **SqlGetTime%** to learn the current timeout value.

## See Also

[SqlExec%](#), [SqlGetTime%](#), [SqlOk%](#), [SqlRpcExec%](#), [SqlSend%](#), [SqlSetLoginTime%](#)

# SqlExec%

Sends the statements in the command buffer to SQL Server for execution.

## Syntax

**SqlExec%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0). Any of the following conditions causes **SqlExec%** to return FAIL:

- The command buffer contains a Transact-SQL syntax error.
- The statements cause a SQL Server permission violation.
- Previous results are not completely processed.
- No statement is specified (the command buffer is empty).
- The network connection is broken.

## Remarks

**SqlExec%** sends Transact-SQL statements stored in the command buffer to SQL Server. You can add statements to the command buffer by calling **SqlCmd%**. If one or more of the statements sent to SQL Server cannot be executed, none of the statements is processed.

Once **SqlExec%** returns SUCCEED, you must call **SqlResults%** to process the results.

**SqlExec%** is equivalent to **SqlSend%** followed by **SqlOk%**.

## Example

The following code fragment shows a typical sequence of calls:

```
'Put the statement into the command buffer.
Result% = SqlCmd%(Sqlconn%, "SELECT data FROM table")

'Send the statement to SQL Server and start execution.
Result% = SqlExec%(Sqlconn%)

'Process the statement results.
Result% = SqlResults%(Sqlconn%)

'Retrieve and process the data in each row.
DO UNTIL SqlNextRow%(Sqlconn%) = NOMOREROWS

    'Code to print or process row of data.

LOOP
```

## See Also

[SqlCmd%](#), [SqlNextRow%](#), [SqlOk%](#), [SqlResults%](#), [SqlSend%](#), [SqlSetTime%](#)

# SqlOk%

Verifies the correctness of a command batch.

## Syntax

**SqlOk%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0). The most common reason that a statement fails is a Transact-SQL syntax error.

## Remarks

**SqlOk%** following **SqlSend%** is the equivalent of **SqlExec%**. **SqlOk%** must be called after **SqlSend%** returns SUCCEED. When **SqlOk%** returns SUCCEED, **SqlResults%** can be called to process the results. **SqlOk%** is useful in text-update operations. When text is sent to SQL Server using **SqlMoreText%**, **SqlOk%** must be called before the first call to **SqlMoreText%** and after the last call to **SqlMoreText%**. For an example of using **SqlOk%** in this context, see "**SqlWriteText%**."

**Note** This function is one of the four that do not return control to the application until the server sends the required response. The application can be blocked for a considerable time if the server is waiting for a lock or processing a large sort. If this is unacceptable, always call **SqlDataReady%** before **SqlOk%** and set the DB-Library timeout to regain control periodically.

## See Also

[SqlCmd%](#), [SqlExec%](#), [SqlNextRow%](#), [SqlResults%](#), [SqlSend%](#), [SqlSetTime%](#)

# SqlSend%

Sends statements in the command buffer to SQL Server and does not wait for a response.

## Syntax

**SqlSend%** ( *sqlconn%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlSend%** sends Transact-SQL statements stored in the command buffer to SQL Server. You can add statements to the command buffer by calling **SqlCmd%**. Once **SqlSend%** returns SUCCEED, you must call **SqlOk%** to verify the accuracy of the statements in the command buffer. Then call **SqlResults%** to process the results.

## See Also

[SqlCmd%](#), [SqlExec%](#), [SqlNextRow%](#), [SqlOk%](#), [SqlResults%](#), [SqlSetTime%](#)

# SqlStrCpy%

Copies a portion of the command buffer to a program variable.

## Syntax

**SqlStrCpy%** ( *sqlconn%*, *start%*, *numbytes%*, *buffer\$* ) ( )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*start%*

Is the character in the command buffer to start copying from. The first character is 0. When *start%* is greater than the length of the command buffer, the returned string is set to an empty string.

*numbytes%* ( )

Is the number of characters to copy.

***numbytes%* setting**

< 0

= 0

> 0

**Result**

**SqlStrCpy%** copies the entire command buffer.

*cmdstring\$* is set to an empty string.

**SqlStrCpy%** copies the actual number of bytes in the buffer and returns SUCCEED.

*buffer\$*

Is a string variable to which the source string is copied.

## Returns

SUCCEED (1) or FAIL (0). FAIL is returned if *start%* is negative.

## Remarks

Internally, the command buffer is a linked list of text strings. Parts of the command buffer can be located and copied using **SqlStrCpy%** and **SqlStrLen%**.

## See Also

[SqlCmd%](#), [SqlFreeBuf](#), [SqlStrLen%](#)

# SqlStrLen%

Returns the length, in characters, of the command buffer.

## Syntax

**SqlStrLen%** ( *sqlconn%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The length, in characters, of the command buffer.

## Remarks

Internally, the command buffer is a linked list of text strings. Parts of the command buffer can be located and copied using **SqlStrLen%** and **SqlStrCpy%**.

## See Also

[SqlCmd%](#), [SqlFreeBuf](#), [SqlStrCpy%](#)

# SqlUse%

Sets the current database for a particular SQL Server connection.

## Syntax

**SqlUse%** ( *sqlconn%*, *sqlname\$* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*sqlname\$*

Is a string containing the database name.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlUse%** sets up the Transact-SQL USE statement and calls **SqlExec%** and **SqlResults%**. If the USE statement fails because the requested database has not yet completed a recovery process, **SqlUse%** continues to send USE statements at 1-second intervals until it succeeds or it encounters some other error.

Before **SqlUse%** sets the current database, it calls **SqlFreeBuf** to clear the command buffer. If you call **SqlCmd%** to place statements into the command buffer, execute these statements before calling **SqlUse%**.

## See Also

[SqlExec%](#), [SqlResults%](#)

# SqlWinExit

Informs DB-Library for Visual Basic that the application is about to exit.

## Syntax

**SqlWinExit**

## Remarks

DB-Library for Visual Basic creates and maintains information about each application that has referenced it to prevent conflicts between applications that use DB-Library for Visual Basic concurrently. In order for DB-Library for Visual Basic to release this information, the application must call **SqlWinExit** just before it exits. Put the call to **SqlWinExit** before you call the **End** function to close an application. For example:

```
Sub Quit_Click()  
    SqlExit  
    SqlWinExit  
End  
End Sub
```

**SqlWinExit** releases the memory DB-Library for Visual Basic allocated to keep track of this application and makes that memory available to other applications.

**Important** Once your application calls **SqlWinExit**, it cannot call any other DB-Library for Visual Basic function. If you call **SqlWinExit** and then want to issue one or more DB-Library for Visual Basic calls, you must again call **SqlInit\$** to re-register your application.

## See Also

[SqlInit\\$](#)



# Cursor Functions

[SqlCursor%](#)

[SqlCursorClose](#)

[SqlCursorCollInfo%](#)

[SqlCursorData\\$](#)

[SqlCursorFetch%](#)

[SqlCursorFetchEx%](#)

[SqlCursorInfo%](#)

[SqlCursorInfoEx%](#)

[SqlCursorOpen%](#)

# SqlCursor%

Inserts, updates, deletes, locks, or refreshes a particular row in the fetch buffer of a client cursor, a transparent server cursor, or an explicit server cursor.

## Syntax

**SqlCursor%** ( *cursorhandle%*, *optype%*, *row%*, *table\$*, *values\$* )

where

*cursorhandle%*

Is the cursor handle previously returned by **SqlCursorOpen%**.

*optype%*

Specifies the type of cursor operation to perform on a row or rows in the fetch buffer, as follows:

<b><i>optype%</i></b>	<b>Description</b>
CRSDELETE%	Deletes row(s).
CRSINSERT%	Inserts a single row using data specified in <i>values\$</i> .
CRSLOCKCCC%	Locks row(s).
	<b>Client cursor:</b>
	An exclusive lock is placed on the data page that contains the specified <i>row%</i> . The lock is maintained only if it is inside an open transaction block defined by BEGIN TRANSACTION; the lock is released when the transaction is closed by a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement.
	<b>Transparent server cursor, explicit server cursor:</b>
	An update intent lock is placed on the data page that contains the specified <i>row%</i> . If not inside an open transaction, this lock is released when the next fetch is performed. If inside an open transaction, this lock is released when the transaction is closed.
CRSREFRESH%	Refreshes row(s) by retrieving current row data from SQL Server.
CRSUPDATE%	Updates row(s) using data specified in <i>values\$</i> .

If the cursor was opened using a *concurop%* of CURREADONLY% (specified in **SqlCursorOpen%**), only CRSREFRESH% is valid.

*row%*

Is the row number in the fetch buffer to which the *optype%* operation applies. The first row in the buffer is number 1. The specified row must contain valid row data.

**Client cursor, transparent server cursor:**

When *optype%* is CRSREFRESH%, a *row%* value of 0 indicates that all rows in the fetch buffer will be refreshed.

**Explicit server cursor:**

When *optype%* is one of the following values:

- CRSDELETE%
- CRSLOCKCCC%

- CRSREFRESH%
- CRSUPDATE%

a *row%* value of 0 indicates that the *optype%* operation will be performed on all rows in the fetch buffer.

#### *table\$*

Is the table to which the *optype%* operation applies. If *optype%* is CRSREFRESH% use an empty string. It must be one of the tables specified in the FROM clause of the SELECT statement (specified in **SqlCursorOpen%**) that defines the cursor. If the FROM clause includes only one table, this parameter is not required, and you can specify that table or an empty string.

#### **Client cursor:**

If the FROM clause includes more than one table, this parameter is required unless:

- *optype%* is CRSINSERT% and *values\$* points to a complete INSERT statement
- *optype%* is CRSUPDATE% and *values\$* points to a complete UPDATE statement

#### **Transparent server cursor, explicit server cursor:**

If the FROM clause includes more than one table and the **SqlCursor%** operation is being performed with an ambiguous column name, this parameter is required. If *table\$* is required but not specified, the default is the first table listed in the FROM clause.

#### *values\$*

Is a string that contains a Transact-SQL statement or clause, or an empty string. This parameter specifies the data to be inserted or updated. The following table lists the valid *values\$* parameters for each *optype%*:

<b><i>optype%</i></b>	<b><i>values\$</i></b>
CRSDELETE%	Empty string
CRSINSERT%	Can be one of the following:  A string that contains a complete Transact-SQL INSERT statement that specifies the single row to be inserted, with no WHERE clause. The table specified in the INSERT statement overrides the <i>table\$</i> parameter. The <i>row%</i> parameter is ignored.  A string that contains just the VALUES clause (from an INSERT statement) that specifies the single row to be inserted. The VALUES keyword is optional, but the list of values to be inserted must be surrounded by parentheses. The <i>row%</i> parameter is ignored.
CRSLOCKCC%	Empty string
CRSREFRESH%	Empty string
CRSUPDATE%	Can be one of the following:  A string that contains a complete UPDATE statement that specifies the changes made to a single row, with no WHERE clause. The table specified in the UPDATE statement overrides the <i>table\$</i> parameter. The <i>row%</i> parameter is ignored.  A string that contains just the SET clause (from an UPDATE statement) that specifies the changes made to a single row. The SET keyword is optional. The <i>row%</i> parameter is ignored.

## **Returns**

SUCCEED (1) or FAIL (0).

This function can fail for the following reasons:

- The cursor is opened as read only, no updates allowed.
- A server or connection failure or timeout occurs.
- You have not been granted permission to update or change the database.
- DB-Library is out of memory.
- A trigger in the database caused the INSERT, LOCK, or UPDATE operation to fail.
- You are using optimistic concurrency control, and the row has changed.

### Remarks

Using **SqlCursor%** does not affect the current cursor position.

When *optype%* is CRSUPDATE% and the *values\$* parameter is a string that contains the UPDATE statement or SET clause, the newly updated values are automatically available using **SqlCursorData\$**.

When using CRSUPDATE%, if a change is made to a column that is part of the unique index used to open the cursor, the changed row will:

- Be missing from a keyset cursor. The next time the changed row is fetched, the row status indicator (*pstatus&()* in **SqlCursorOpen%**) for that row will be FTCMISSING%.
- Appear in a new position in a dynamic cursor. The new position depends on the new value of the unique index column, and later fetches might retrieve the changed row.

After using CRSDELETE%, deleted rows will be missing from a keyset cursor (later fetches will have a row status of FTCMISSING%), and will disappear from later fetches using dynamic cursors.

### Client cursor:

When using CRSINSERT% with a keyset cursor, the inserted row does not appear in the cursor results set, and thus does not appear in later fetches.

### Transparent server cursor, explicit server cursor:

When *optype%* is one of the following:

- CRSINSERT% with a keyset cursor involving only one table
- CRSUPDATE% to change a column that is part of the unique index used to open the cursor

the inserted or updated row will appear as a new row at the end of the keyset (even if the inserted row does not match the WHERE clause criteria), or it will appear in the position of a missing row if the unique index columns of the inserted or updated row match the unique index columns of the missing row.

### See Also

[SqlCursorClose](#), [SqlCursorCollInfo%](#), [SqlCursorFetch%](#), [SqlCursorInfo%](#), [SqlCursorOpen%](#)

# SqlCursorClose

Closes the client cursor, transparent server cursor, or explicit server cursor, and releases the memory associated with the cursor handle.

## Syntax

**SqlCursorClose** ( *cursorhandle%* )

where

*cursorhandle%*

Is a cursor handle returned by **SqlCursorOpen%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

After issuing **SqlCursorClose**, the cursor handle should not used.

## See Also

[SqlCursor%](#), [SqlCursorCollInfo%](#), [SqlCursorFetch%](#), [SqlCursorInfo%](#), [SqlCursorOpen%](#)

# SqlCursorCollInfo%

Returns information about the specified column of a client cursor, transparent server cursor, or explicit server cursor.

## Syntax

**SqlCursorCollInfo%** ( *cursorhandle%*, *column%*, *colname\$*, *coltype%*, *collen&*, *usertype%* )

where

*cursorhandle%*

Is the cursor handle created by **SqlCursorOpen%**.

*column%*

Is the column number in the cursor for which information is to be returned. The first column is number 1.

*colname\$*

Is a string that will contain the name of the column. The program variable should be large enough to accommodate the column name.

*coltype%*

Is a program variable that will contain the datatype token of the column. If *coltype%* is set to -1, the column datatype is not returned.

*collen&*

Is a program variable that will contain the maximum length of the column, in bytes. If *collen&* is set to -1, the maximum column length is not returned.

*usertype%*

Is a program variable that will contain the user-defined datatype of the column. If *usertype%* is set to -1, the column user-defined datatype is not returned.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

Call **SqlCursorCollInfo%** after **SqlCursorOpen%** returns a valid cursor handle. The **SqlCollInfo%** function returns more detailed information about a cursor column.

## See Also

[SqlCollInfo%](#), [SqlCursor%](#), [SqlCursorClose](#), [SqlCursorFetch%](#), [SqlCursorInfo%](#), [SqlCursorOpen%](#)

# SqlCursorData\$

Returns the value of the data in a specified row and column of the current fetch buffer for a client cursor, a transparent server cursor, or an explicit server cursor.

## Syntax

**SqlCursorData\$** ( *cursorhandle%*, *rownum%*, *column%* )

where

*cursorhandle%*

Is the cursor handle created by **SqlCursorOpen%**.

*rownum%*

Is the number of the row in the current fetch buffer to retrieve data from. The first row is number 1.

*column%*

Is the number of the column in the current fetch buffer to retrieve data from. The first column is number 1.

## Returns

The value of the data in the specified row and column. When the column is NULL or the *rownum%* or *column%* is out of range, an empty string is returned.

## Remarks

To convert the data from a string to a different datatype, use **SqlColInfo%** to get the datatype of the data, and then use Visual Basic functions to convert to Visual Basic datatypes. Some helpful Visual Basic conversion functions are shown in the following table. For more information, see your documentation for Visual Basic.

Visual Basic function	Description
CDBL	Converts a numeric expression to a double-precision value.
CLNG	Converts a numeric expression to a long integer.
VAL	Converts a string representation of a value to a numeric expression.

## See Also

[SqlColType%](#), [SqlCursor%](#), [SqlCursorClose](#), [SqlCursorColInfo%](#), [SqlCursorFetch%](#), [SqlCursorInfo%](#), [SqlCursorOpen%](#)

# SqlCursorFetch%

Fetches a block of rows (called the fetch buffer) from a client cursor or transparent server cursor, and makes the rows available using **SqlCursorData\$**. If you are connected to SQL Server 6.0, you should use **SqlCursorFetchEx%**.

## Syntax

**SqlCursorFetch%** ( *cursorhandle%*, *fetchtype%*, *rownum%* )

where

*cursorhandle%*

Is the cursor handle created by **SqlCursorOpen%**.

*fetchtype%*

Specifies the type of fetch to execute, changing the position of the fetch buffer within the cursor results set. The following table describes the different *fetchtype%* values:

<i>fetchtype%</i>	Description
FETCHFIRST%	Fetches the first block of rows from a dynamic or keyset cursor. The first row of the new fetch buffer is the first row in the cursor results set.
FETCHNEXT%	Fetches the next block of rows from a dynamic or keyset cursor. The first row of the new fetch buffer is the row after the last row of the current fetch buffer.  If this is the first fetch using a new cursor, it behaves the same as FETCHFIRST%.
FETCHPREV%	Fetches the previous block of rows from a fully dynamic or keyset cursor. The first row of the new fetch buffer is <i>nrows%</i> (specified in <b>SqlCursorOpen%</b> ) before the first row of the current fetch buffer.
FETCHRANDOM%	Fetches a block of rows from a keyset cursor. The first row of the new fetch buffer is the specified <i>rownum%</i> row in the keyset cursor results set.
FETCHRELATIVE%	Fetches a block of rows from a keyset cursor. The first row of the new fetch buffer is <i>rownum%</i> rows before or after the first row of the current fetch buffer.
FETCHLAST%	Fetches the last block of rows from a keyset cursor. The last row of the new fetch buffer is the last row of the cursor results set.

The block of rows retrieved by a fetch is called the fetch buffer. The number of rows in the fetch buffer is determined by the *nrows%* parameter of **SqlCursorOpen%**.

For a forward-only dynamic cursor (*scrollopt%* is CURFORWARD% in **SqlCursorOpen%**), you can use only FETCHFIRST% or FETCHNEXT%.

*rownum%*

Is the specified random or relative row number to use as the first row of the new fetch buffer. Use this parameter only with a *fetchtype%* of FETCHRANDOM% or FETCHRELATIVE%. Specify 0 for any other *fetchtype%*.

When *fetchtype%* is FETCHRANDOM%, the first row of the new fetch buffer is the *rownum%* row (counting forward from the beginning) of the keyset cursor results set. The *rownum%* parameter



must be positive.

When *fetchtype%* is FETCHRELATIVE%:

- A positive *rownum%* means that the first row of the new fetch buffer is *rownum%* rows after the first row of the current fetch buffer.
- A negative *rownum%* means that the first row of the new fetch buffer is *rownum%* rows before the first row of the current fetch buffer.
- A *rownum%* of 0 means that all rows in the fetch buffer are refreshed with current data from SQL Server without moving the current cursor position. This is identical to calling **SqlCursor%** with *optype%* set to CRSREFRESH%.

## Returns

SUCCEED (1) or FAIL (0).

SUCCEED (1) is returned if every row was fetched successfully. Note that for a keyset cursor, a fetch that results in a missing row will not cause **SqlCursorFetch%** to FAIL (0).

FAIL (0) is returned if at least one of the following is true:

- A *fetchtype%* of FETCHRANDOM%, FETCHRELATIVE%, or FETCHLAST% was used on a dynamic cursor.
- A *fetchtype* other than FETCHFIRST% or FETCHNEXT% was used on a forward-only dynamic cursor.
- The SQL Server connection is broken or times out.
- DB-Library is out of memory.

## Remarks

Specify the size of the fetch buffer in the *nrows%* parameter of **SqlCursorOpen%**.

After the fetch, the elements of the array of row status indicators (*pstatus&()* in **SqlCursorOpen%**) are filled with row status values, one for each row in the fetch buffer. Each row status value is a series of fetch status values ORed together. The following table shows the meaning of each row status value:

Fetch status	Description
FTCSUCCEED%	The row was successfully fetched. <b>SqlCursorData\$</b> will return valid data for the row.
FTCMISSING%	The row has been deleted or a unique index column of the row has been changed. Do not use <b>SqlCursorData\$</b> for the row.  For keyset cursors, this fetch status can appear at any time. For dynamic cursors, this fetch status can appear only after the current fetch buffer is refreshed.
FTCENDOFKEYSET%	Is the end of the keyset. This fetch status is set for backward compatibility with "mixed" client cursors used by existing applications.
FTCENDOFRESULTS%	Is the end of the results set of a dynamic or keyset cursor. Rows in the fetch buffer after this row are invalid and will have a row status indicator of 0; do not use <b>SqlCursorData\$</b> for those rows.  If ORed with FTCSUCCEED%, this is the last row in the cursor results set; it contains valid

data.

If ORed with FTCSUCCESS%, this is the last row in the cursor results set, but the row is missing.

If not ORed with FTCSUCCESS% or FTCSUCCESS%, this row is invalid.

A row status indicator of 0 means that the row is invalid, and **SqlCursorData\$** will not return valid data. This usually happens when the row is before the beginning (first row) or after the end (last row) of the cursor results set.

After the fetch, **SqlCursorData\$** returns:

- Valid data for all rows with a fetch status of FTCSUCCESS%.
- Invalid data for all rows *without* a fetch status of FTCSUCCESS%.

If no fetches have been performed on a cursor, the current cursor position is before the beginning (first row) of the cursor results set.

After a fetch is complete, the new cursor position is one of the following:

- The first row of the new fetch buffer, as specified under *fetchtype%*, if the first row of the new fetch buffer stayed within the cursor results set.
- Adjusted to the first row of the cursor results set if the first row of the new fetch buffer would have been before the first row of the cursor results set and the last row of the new fetch buffer would have stayed within the cursor results set due to a FETCHPREV% operation.
- Unchanged from the current position if all rows of the new fetch buffer would have been before the first row of the cursor results set, if the first row (and thus all rows) of the new fetch buffer would have been after the last row of the cursor results set, or, if for a keyset cursor, if the first row of the new fetch buffer would have been before the first row of the cursor results set and the last row of the new fetch buffer would have stayed within the cursor results set due to a FETCHRELATIVE% attempt with a negative *rownum%*.

When the new cursor position is unchanged because the first row (and thus all rows) of the new fetch buffer would have been after the last row of the cursor results set, all rows in the fetch buffer are invalid and will not have a fetch status of FTCSUCCESS%. In the case of dynamic cursors, the first row of the fetch buffer will have a fetch status of FTENDOFRESULTS%, and later rows will have a row status of 0. In the case of keyset cursors, all rows in the fetch buffer will have a row status of 0.

When the new position of a dynamic cursor is unchanged because all rows of the new fetch buffer would have been before the first row of the cursor results set, all rows in the fetch buffer are invalid and will not have a fetch status of FTCSUCCESS%. The first row of the fetch buffer will have a fetch status of FTENDOFRESULTS%, and later rows will have a row status of 0.

When the new cursor position is unchanged and all rows in the fetch buffer are invalid, you can use **SqlCursor%** to refresh the rows in the fetch buffer with current data from SQL Server. This will result in valid rows that reflect the current cursor position.

Each call to **SqlCursorFetch%** leaves the connection available for use with no pending results.

**Note** This function works with client cursors and transparent server cursors. Do not use both **SqlCursorFetch%** and **SqlCursorFetchEx%** with the same cursor handle. Once one of these functions is used on a specific cursor handle, any attempt to use the other function will return FAIL (0).

If rows in the current fetch buffer of a dynamic cursor are deleted, a fetch using a client cursor might behave differently than a fetch using a transparent server cursor.

**Client cursor:**

When the new position of a dynamic cursor is adjusted to be the first row of the cursor results set (which happens when the first row of the new fetch buffer would have been before the first row of the dynamic cursor results set and the last row of the new fetch buffer would have stayed within the dynamic cursor results set due to a FETCHPREV% operation), some rows at the end of the new fetch buffer might be invalid. Any invalid rows will have a row status indicator of 0.

If rows in the current fetch buffer of a dynamic cursor are deleted, a fetch next or fetch previous might result in a new fetch buffer that skips rows in the cursor results set or includes rows from the current fetch buffer again.

**Transparent server cursor:**

A *fetchtype%* of FETCHNEXT% or FETCHPREV% using a dynamic cursor is actually mapped to a relative fetch on SQL Server 6.0. Because of this, if the first row in the current fetch buffer is deleted before a FETCHNEXT% (mapped to a forward relative fetch on SQL Server 6.0) is performed, the current cursor position becomes invalid. For more information about the fetch behavior in this case, see **SqlCursorFetchEx%**.

**See Also**

[SqlCursor%](#), [SqlCursorCollInfo%](#), [SqlCursorClose](#), [SqlCursorInfo%](#), [SqlCursorOpen%](#)

# SqlCursorFetchEx%

Fetches a block of rows (called the fetch buffer) from an explicit server cursor and makes the rows available using **SqlCursorData\$**.

## Syntax

**SqlCursorFetchEx%** ( *cursorhandle%*, *fetchtype%*, *rownum%*, *nfetchrows%*, *reserved&* )

where

*cursorhandle%*

Is an explicit server cursor handle returned by **SqlCursorOpen%**.

*fetchtype%*

Specifies the type of fetch to execute, changing the position of the fetch buffer within the cursor results set. The following table describes the different *fetchtype%* values:

<i>fetchtype%</i>	Description
FETCHFIRST%	Fetches the first block of rows from a dynamic or keyset cursor. The first row of the new fetch buffer is the first row in the cursor results set.
FETCHNEXT%	Fetches the next block of rows from a dynamic or keyset cursor. The first row of the new fetch buffer is the row after the last row of the current fetch buffer.  If this is the first fetch using a new cursor, it behaves the same as FETCHFIRST%.
FETCHPREV%	Fetches the previous block of rows from a fully dynamic or keyset cursor. The first row of the new fetch buffer is <i>nrows%</i> (specified in <b>SqlCursorOpen%</b> ) before the first row of the current fetch buffer.
FETCHRANDOM%	Fetches a block of rows from a keyset cursor. The first row of the new fetch buffer is the specified <i>rownum%</i> row in the cursor results set.
FETCHRELATIVE%	Fetches a block of rows from a dynamic or keyset cursor. The first row of the new fetch buffer is <i>rownum%</i> rows before or after the first row of the current fetch buffer.
FETCHLAST%	Fetches the last block of rows from a dynamic or keyset cursor. The last row of the new fetch buffer is the last row of the cursor results set.

The block of rows retrieved by a fetch is called the fetch buffer. The number of rows in the fetch buffer is determined by the *nfetchrows%* parameter.

For a forward-only dynamic cursor (*scrollopt%* is CURFORWARD% in **SqlCursorOpen%**), you can only use the FETCHFIRST%, FETCHNEXT%, or FETCHRELATIVE% (with a positive *rownum%*) types.

*rownum%*

Is the specified random or relative row number to use as the first row of the new fetch buffer. Use this parameter only with a *fetchtype%* of FETCHRANDOM% or FETCHRELATIVE%. Specify 0 for any other *fetchtype%*.

When *fetchtype%* is FETCHRANDOM%:

- A positive *rownum%* means that the first row of the new fetch buffer is the *rownum%* row (counting forward from the beginning) of the cursor results set.
- A negative *rownum%* means that the first row of the new fetch buffer is *rownum%* rows backward from the end of the cursor results set. Given N rows in the cursor results set, the first row of the new fetch buffer is row  $N+1+rownum\%$  of the cursor results set.  
For example, a *rownum%* of - 1 means the first row of the new fetch buffer is row N ( $N+1 - 1$ ), or the last row, of the current results set. A *rownum%* of - N means the first row of the new fetch buffer is row 1 ( $N+1 - N$ ), or the first row, of the current results set.
- A *rownum%* of 0 means that the first row of the new fetch buffer is before the beginning (first row) of the cursor results set.

When *fetchtype%* is FETCHRELATIVE%:

- A positive *rownum%* means that the first row of the new fetch buffer is *rownum%* rows after the first row of the current fetch buffer.  
For dynamic cursors, if the first row in the current fetch buffer is deleted before a relative fetch, the current cursor position becomes invalid. Let D be the number of contiguous rows, including the first row, deleted from the beginning of the current fetch buffer. Before executing a relative fetch, the current cursor position is set to before the first non-deleted row (row D+1) in the current fetch buffer.  
In this case, when a relative fetch is performed with a positive *rownum%*, the first row of the new fetch buffer is row  $rownum\%+D$  of the current fetch buffer.
- A negative *rownum%* means that the first row of the new fetch buffer is *rownum%* rows before the first row of the current fetch buffer.

- For dynamic cursors, a *rownum%* of 0 means that all the rows in the current fetch buffer are fetched again without moving the current cursor position. This is different from a refresh because the rows in the new fetch buffer can differ from the rows in the current fetch buffer. New rows can appear, and old rows can disappear.

In the case given above, after D contiguous rows have been deleted from the beginning of the current fetch buffer, when a relative fetch is performed with a *rownum%* of 0, the first row of the new fetch buffer is the first non-deleted row (row D+1) of the current fetch buffer.

For keyset cursors, a *rownum%* of 0 means that the current fetch buffer is refreshed with current data from SQL Server without moving the current cursor position. This is identical to calling **SqlCursor%** with *optype%* set to CRSREFRESH%.

#### *nfetchrows%*

Is the number of rows in the new fetch buffer. This value must be less than or equal to the *nrows%* parameter specified for this cursor in **SqlCursorOpen%**. The *poutlen* and *pvaraddr* arrays specified in calls to **dbcursorbind** must have at least *nfetchrows%* elements. If these arrays are not large enough, you must break the existing bindings and then rebind with large enough arrays (at least *nfetchrows%* elements) before calling **SqlCursorFetchEx%**.

When *fetchtype%* is FETCHFIRST%, an *nfetchrows%* value of 0 means that the new cursor position is set to before the beginning (first row) of the cursor results set.

When *fetchtype%* is FETCHLAST%, an *nfetchrows%* value of 0 means that the new cursor position is set to after the end (last row) of the cursor results set.

#### *reserved&*

Reserved for future use. Use 0.

### Returns

SUCCEED (1) or FAIL (0).

SUCCEED (1) is returned if every row was fetched successfully. Note that for a keyset cursor, a fetch that results in a missing row will not cause **SqlCursorFetchEx%** to FAIL (0).

FAIL (0) is returned if at least one of the following is true:

- A *fetchtype%* of FETCHRANDOM% was used on a dynamic cursor.
- A *fetchtype* other than FETCHFIRST%, FETCHNEXT%, or FETCHRELATIVE% (with a positive *rownum%*) was used on a forward-only dynamic cursor.
- The SQL Server connection is broken or times out.
- DB-Library is out of memory.

### Remarks

After the fetch, the elements of the array of row status indicators (*pstatus&()* in **SqlCursorOpen%**) are filled with row status values, one for each row in the fetch buffer. Each row status value is a series of fetch status values ORed together. The following table shows the meaning of each row status value:

Fetch status	Description
FTCSUCCEED%	The row was successfully fetched. <b>SqlCursorData\$</b> will return valid data for the row.
FTCMISSING%	The row has been deleted or a unique index column of the row has been changed. Do not use the values returned by <b>SqlCursorData\$</b> for the row.  For keyset cursors, this fetch status can appear at any time. For dynamic cursors, this fetch status can appear only after the current fetch buffer is refreshed.

A row status indicator of 0 means that the row is invalid, and **SqlCursorData\$** will not return valid data. This happens when the row is before the beginning (first row) or after the end (last row) of the cursor results set.

After the fetch, **SqlCursorData\$** returns:

- Valid data for all rows with a fetch status of FTCSUCCEED%.
- Invalid data for all rows *without* a fetch status of FTCSUCCEED%.

If no fetches have been performed on a cursor, the current cursor position is before the beginning (first row) of the cursor results set.

After a fetch is complete, the new explicit server cursor position is one of the following:

- The first row of the new fetch buffer, as specified under *fetchtype%*, if the first row of the new fetch buffer stayed within the cursor results set.
- Adjusted to the first row of the cursor results set if the first row of the new fetch buffer would have been before the first row of the cursor results set and the last row of the new fetch buffer would have stayed within the cursor results set.
- Before the beginning of the cursor results set if all rows of the new fetch buffer are before the first row of the cursor results set, or if any backward fetch (FETCHPREV% or FETCHRELATIVE% with a negative *rownum%*) is performed when the first row of the current fetch buffer is the first row of the cursor results set.
- After the end of the cursor results set if the first row (and thus all rows) of the new fetch buffer is after the last row of the cursor results set.

When the current cursor position is before the beginning of the cursor, a FETCHNEXT% operation is identical to a FETCHFIRST% operation. When the current cursor position is after the end of the cursor, a FETCHPREV% operation is identical to a FETCHLAST% operation.

**Note** This function works with explicit server cursors in SQL Server 6.0. Do not use both

**SqlCursorFetchEx%** and **SqlCursorFetch%** with the same server cursor handle. Once one of these functions is used on a specific cursor handle, any attempt to use the other function will return FAIL (0).

Each call to **SqlCursorFetch%** leaves the connection available for use with no pending results.

**See Also**

[SqlCursor%](#), [SqlCursorCollInfo%](#), [SqlCursorClose](#), [SqlCursorInfo%](#), [SqlCursorOpen%](#)

# SqlCursorInfo%

Returns the number of columns and the number of rows in the keyset for a client cursor, a transparent server cursor, or an explicit server cursor. The **SqlCursorInfoEx%** function returns more detailed information.

## Syntax

**SqlCursorInfo%** ( *cursorhandle%*, *ncols%*, *nrows&* )

where

*cursorhandle%*

Is the cursor handle created by **SqlCursorOpen%**.

*ncols%*

Is a program variable that will contain the number of columns in the cursor.

*nrows&*

Is a program variable that will contain the number of rows in a cursor results set.

### Client cursor:

For a keyset cursor, this number is always valid. For a dynamic cursor, this number is valid only if the current fetch buffer contains the last row in the cursor results set; otherwise, - 1 is returned.

### Transparent server cursor, explicit server cursor:

For a dynamic cursor, - 1 is returned.

For a keyset cursor, this value can be the number of rows populated if asynchronous population of the cursor results set is incomplete, or the total number of rows in the cursor results set. You can call **SqlCursorInfoEx%** to determine this.

## Returns

SUCCESS (1) or FAIL (0).

## Remarks

Call **SqlCursorInfoEx%** for more complete information about transparent server cursors and explicit server cursors.

## See Also

[SqlCursor%](#), [SqlCursorClose](#), [SqlCursorColInfo%](#), [SqlCursorFetch%](#), [SqlCursorInfoEx%](#), [SqlCursorOpen%](#)



# SqlCursorInfoEx%

Returns information about a client cursor, a transparent server cursor, or an explicit server cursor.

## Syntax

**SqlCursorInfoEx%**( *cursorhandle%*, *cursorinfo* )

where

*cursorhandle%*

Is a cursor handle returned by **SqlCursorOpen%**.

*cursorinfo*

Is a *SqlCursorInfo* structure that DB-Library will fill with information about the specified cursor.

The *SqlCursorInfo* structure is defined as follows:

```
Type SqlCursorInfo
    TotCols As Long
    TotRows As Long
    CurRow As Long
    TotRowsFetched As Long
    CursorType As Long
    Status As Long
End Type
```

The *SqlCursorInfo* fields are described below:

Field	Description
<b>TotCols</b>	Is the total number of columns in the cursor.
<b>TotRows</b>	<p>Is the total number of rows in the cursor results set.</p> <p><b>Client cursor:</b></p> <p>For a keyset cursor, this number is always valid. For a dynamic cursor, this number is valid only if the current fetch buffer contains the last row in the cursor results set; otherwise, - 1 is returned.</p> <p><b>Transparent server cursor, explicit server cursor:</b></p> <p>For a dynamic cursor, - 1 is returned.</p> <p>For a keyset cursor, if the <b>Status</b> field is CU_FILLING, the asynchronous population of the cursor results set is incomplete, and this value indicates the number of rows populated. If the <b>Status</b> field is CU_FILLED, the cursor results set is completely populated, and this value indicates the total number of rows in the cursor results set.</p>
<b>CurRow</b>	<p>Is the row number within the cursor results set of the current cursor position (first row of the fetch buffer). The first row of the cursor results set is number 1.</p> <p><b>Client cursor:</b></p> <p>This value will be 0.</p>

	<p><b>Transparent server cursor, explicit server cursor:</b></p> <p>For a keyset cursor, this value is always valid.</p> <p>For a dynamic cursor, this value will always be 1 when the current position is within the cursor results set.</p> <p>This value will be 0 if the current position is before the beginning of the cursor. This value will be - 1 if the current position is after the end of the cursor.</p>
<b>TotRowsFetched</b>	Is the total number of valid rows in the current fetch buffer.
<b>Type</b>	<p>Is a bitmap of cursor type, scroll option, and concurrency control information. It is a series of the following values ORed together:</p> <p><b>Type:</b></p> <p>CU_CLIENT: Client cursor</p> <p>CU_SERVER: Transparent server cursor or explicit server cursor</p> <p><b>Scroll option:</b></p> <p>CU_DYNAMIC: Dynamic cursor</p> <p>CU_FORWARD: Forward-only dynamic cursor</p> <p>CU_KEYSET: Keyset cursor</p> <p>CU_INSENSITIVE: Insensitive keyset cursor</p> <p>CU_MIXED: Mixed-mode cursor (provided for backward compatibility only)</p> <p><b>Concurrency control:</b></p> <p>CU_READONLY: Read-only concurrency</p> <p>CU_LOCKCC: Intent to update concurrency</p> <p>CU_OPTCC: Optimistic concurrency based on timestamp or values</p> <p>CU_OPTCCVAL: Optimistic concurrency based on values</p>
<b>Status</b>	<p>Bitmap of status information. It is a series of the following values ORed together:</p> <p><b>Client cursor:</b></p> <p>CU_FILLED: All cursors</p> <p><b>Transparent server cursor, explicit server cursor:</b></p> <p>CU_FILLING: Incomplete asynchronous population of a keyset transparent server cursor or keyset explicit server cursor results set</p> <p>CU_FILLED: Incomplete asynchronous population of a keyset transparent server cursor or keyset explicit server cursor results set, or the cursor is a dynamic cursor</p>

**Returns**

SUCCEED (1) or FAIL (0).

**Remarks**

Before calling **SqlCursorInfoEx%**, set the **SizeOfStruct** field equal to the value returned by the C **sizeof** function for the SQLCURSORINFO% structure. The **SqlCursorInfoEx%** function fills the supplied SQLCURSORINFO% structure with information about the open server cursor.

**See Also**

[SqlCursor%](#), [SqlCursorClose](#), [SqlCursorCollInfo%](#), [SqlCursorFetch%](#), [SqlCursorInfoEx%](#), [SqlCursorOpen%](#)

# SqlCursorOpen%

Opens a cursor. It is an explicit server cursor if you are connected to SQL Server 6.0 and **SqlCursorFetchEx%** is used for the first fetch. It is a transparent server cursor if you are connected to SQL Server 6.0 and **SqlCursorFetch%** is used for the first fetch. It is a client cursor if you are connected to SQL Server 4.2, or if the DBCLIENTCURSORS option is set, and only **SqlCursorFetch%** can be used to fetch rows.

## Syntax

**SqlCursorOpen%** ( *sqlconn%*, *stmt\$*, *scrollopt%*, *concuropt%*, *nrows%*, *pstatus&()* )

where

*sqlconn%*

Is the SQL Server connection returned by **SqlOpen%**.

*stmt\$*

Is the SELECT statement that defines a cursor.

### Client cursor:

This must be a single SELECT statement. All tables included in the FROM clause must have a unique index.

The SELECT statement cannot contain any of the following:

- INTO
- FOR BROWSE
- COMPUTE
- UNION
- COMPUTE BY
- Aggregate function
- Table alias

If *scrollopt%* is CURKEYSET%, the SELECT statement can contain the following keywords:

- ORDER BY
- HAVING
- GROUP BY

If the SELECT statement includes a view, the FROM clause must include only a single view (no other tables or views). All base tables included in the FROM clause of the view definition must have a unique index, and the select list must include all unique index columns of the base tables.

### Transparent server cursor, explicit server cursor:

This can be a single SELECT statement or the name of a stored procedure that contains only a single SELECT statement.

The SELECT statement (alone or in a stored procedure) cannot contain any of the following keywords:

- INTO
- FOR BROWSE
- COMPUTE

The SELECT statement can contain an ORDER BY clause. If the columns in the ORDER BY clause match the columns of the unique indexes used by the cursor, the cursor will use the *scrollopt%* requested. If they do not match, SQL Server must generate a temporary table, and a CURKEYSET% cursor will be used if a *scrollopt%* of CURFORWARD% or CURDYNAMIC% is requested. This also occurs if the SELECT contains a subquery.

The cursor is automatically opened with a *scrollopt%* of CURINSENSITIVE% and a *concuropt%* of

CURREADONLY% if the SELECT statement contains any of the following:

- Table with no unique index
- UNION
- DISTINCT
- GROUP BY
- HAVING
- Aggregate function
- Outer join

If a stored procedure is used, any input parameters must be constants. Declared variables cannot be used for input parameters. Any output parameters or return values from the stored procedure are ignored.

*scrollopt%*

Is one of the following requested scroll options:

<b><i>scrollopt%</i></b>	<b>Description</b>
CURDYNAMIC%	Dynamic cursor. <b>Client cursor, transparent server cursor:</b> The <b>SqlCursorFetch%</b> function will allow only a <i>fetchtype%</i> of FETCHFIRST%, FETCHNEXT%, or FETCHPREV%. <b>Explicit server cursor:</b> The <b>SqlCursorFetchEx%</b> function will allow all <i>fetchtype%</i> values except FETCHRANDOM%.
CURFORWARD%	Forward-only dynamic cursor. <b>Client cursor, transparent server cursor:</b> The <b>SqlCursorFetch%</b> function will allow only a <i>fetchtype%</i> of FETCHFIRST% or FETCHNEXT%. <b>Explicit server cursor:</b> The <b>SqlCursorFetchEx%</b> function will allow only a <i>fetchtype%</i> of FETCHFIRST%, FETCHNEXT%, or FETCHRELATIVE% with a positive <i>rownum%</i> .
CURKEYSET%	Keyset cursor. The <b>SqlCursorFetch%</b> and <b>SqlCursorFetchEx%</b> functions will allow all <i>fetchtype%</i> values.
CURINSENSITIVE%	<b>Client cursor:</b> Not supported. <b>Transparent server cursor, explicit server cursor:</b> Insensitive keyset cursor. Use a <i>concurop%</i> of READ_ONLY. SQL Server will generate a temporary table, so changes made to the rows by others will not be visible through the cursor. The <b>SqlCursorFetch%</b> and <b>SqlCursorFetchEx%</b> functions will allow all <i>fetchtype%</i> values.
<i>n &gt; 1</i>	<b>Client cursor:</b> For backward compatibility with "mixed" client cursors. <b>Transparent server cursor, explicit server</b>

**cursor:**

Mapped to a CURKEYSET% cursor.

*concurop%*

Is one of the following concurrency control options:

**concurop%**

**Description**

CURREADONLY  
%

Read-only cursor. You cannot modify rows in the cursor results set.

CURLOCKCC%

Intent to update locking.

**Client cursor:**

Places an exclusive lock on the data page that contains each row as it is fetched. The locks are maintained only if it is inside an open transaction block defined by BEGIN TRANSACTION; the locks are released when the transaction is closed by a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement.

**Transparent server cursor, explicit server cursor:**

Places an update intent lock on the data page that contains each row as it is fetched. If not inside an open transaction, the locks are released when the next fetch is performed. If inside an open transaction, the locks are released when the transaction is closed.

CUROPTCC%

Optimistic concurrency control using timestamp or values. Changes to a row through the cursor succeed only if the row remains unchanged since the last fetch. Changes are detected by comparing timestamps or by comparing all non-text, non-image values if timestamps are not available.

CUROPTCCVAL%

Optimistic concurrency control using values. Changes to a row through the cursor succeed only if the row remains unchanged since the last fetch. Changes are detected by comparing all non-text, non-image values.

*nrows%*

**Client cursor, transparent server cursor:**

Is the number of rows in the fetch buffer filled by calls to SqlCursorFetch%.

**Explicit server cursor:**

Is the maximum number of rows in the fetch buffer. The *nfetchrows%* parameter of SqlCursorFetchEx% must be less than or equal to this value.

*pstatus&()*

Is an array of row status indicators. This array must contain *nrows%* long integer elements. A row status value is a bitmap of fetch status values ORed together. When passing a *pstatus&()* parameter to **SqlCursorOpen%**, pass the first element of the array, for example, **mypstatus&(0)**.

Each row in the fetch buffer has a corresponding row status indicator. After a fetch, the status of every row in the fetch buffer is returned in the corresponding element of this array.

**Client cursor, transparent server cursor:**

For more information about fetch status values, see **SqlCursorFetch%**.

**Explicit server cursor:**

For more information about fetch status values, see **SqlCursorFetchEx%**.

**Returns**

A handle to the cursor if the cursor open succeeds. If it fails, 0 is returned. Several errors, such as the following, can cause the cursor to fail:

- Not enough memory to complete the request. Reduce the number of rows in the keyset cursor results set by using a more limiting WHERE clause, use a dynamic cursor, or reduce the number of rows in the fetch buffer.
- Tables did not have the required unique indexes.
- A syntax error occurred or the SELECT statement failed.

**Remarks**

After **SqlCursorOpen%** returns a valid cursor handle, you can call **SqlCursorInfoEx%** and examine the **Type** field to determine the actual type of cursor that was opened.

With a dynamic cursor, membership of rows in the cursor results set is determined at fetch time, and it can change between each fetch. A row disappears from the cursor results set if it is deleted or if it is updated such that it no longer meets the WHERE clause criteria. A row appears in the cursor results set if it is inserted or updated such that it meets the WHERE clause criteria.

With a keyset cursor, membership and order of rows in the cursor results set is fixed at open time. A row is marked as missing from the cursor results set if it is deleted or if it is updated such that it no longer meets the WHERE clause criteria. A row appears in the cursor results set only if it is inserted through a cursor based on a single table.

Multiple cursors (as many as the system's memory allows) can be opened using the same connection. When cursor functions are called, there should be no commands waiting to be executed or results pending in the connection.

**See Also**

[SqlCursor%](#), [SqlCursorClose](#), [SqlCursorCollInfo%](#), [SqlCursorFetch%](#), [SqlCursorInfo%](#), [SqlCursorOpen%](#)

# Stored Procedure Functions

[SqlHasRetStat%](#)

[SqlNumRets%](#)

[SqlRetData\\$](#)

[SqlRetLen&](#)

[SqlRetName\\$](#)

[SqlRetStatus&](#)

[SqlRetType%](#)

[SqlRpcExec%](#)

[SqlRpcInit%](#)

[SqlRpcParam%](#)

[SqlRpcSend%](#)



# SqlHasRetStat%

Determines whether a stored procedure or a remote stored procedure generated a return status number.

## Syntax

**SqlHasRetStat%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

Status numbers are returned only by stored procedures running on SQL Server version 4.2 or later.

The server returns stored procedure information (including any return status and parameter values) immediately after returning all normal results for that stored procedure. Process the normal results, and then call **SqlHasRetStat%** after **SqlResults%** returns NOMORERPCRESULTS (for all stored procedures in a batch except the last one) or NOMORERESULTS (for a single stored procedure, or for the last stored procedure in a batch).

**SqlRetStatus&** actually retrieves the status number. Stored procedures that complete normally return a status number of 0.

The order in which the application processes the status number and any return parameter values is unimportant.

## See Also

[SqlNextRow%](#), [SqlResults%](#), [SqlRetData\\$](#), [SqlRetStatus&](#)

# SqNumRets%

Calculates the number of returned parameter values generated by a stored procedure or a remote stored procedure.

## Syntax

**SqNumRets%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqOpen%**.

## Returns

The number of parameter values associated with the most recently executed stored procedure.

## Remarks

For more information about stored procedure return parameters, see "**SqRetData\$**."

The server returns stored procedure information (including any return status and parameter values) immediately after returning all normal results for that stored procedure. Process the normal results, and then call **SqNumRets%** after **SqResults%** returns NOMORERPCRESULTS (for all stored procedures in a batch except the last one) or NOMORERESULTS (for a single stored procedure, or for the last stored procedure in a batch).

## See Also

[SqNextRow%](#), [SqResults%](#), [SqRetData\\$](#), [SqRetLen&](#), [SqRetName\\$](#), [SqRetType%](#)

# SqlRetData\$

Returns a return-parameter value generated by a stored procedure or a remote stored procedure.

## Syntax

**SqlRetData\$** ( *sqlconn%*, *retnum%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*retnum%*

Is the number of the returned value of interest. The first return value is number 1. Values are returned in the same order as the parameters were originally specified in the stored procedure's CREATE PROCEDURE statement. (Note that the order is not necessarily the same as that specified in the remote stored procedure.) When specifying *retnum%*, non-return parameters are not counted. For example, if the second parameter in a stored procedure is the only return parameter, its *retnum%* is 1, not 2.

## Returns

A string containing the value of the specified *retnum%*. If *retnum%* is out of range, **SqlRetData\$** returns an empty string. To determine whether the data really has a null value (and *retnum%* is not merely out of range), check for a return value of 0 from **SqlRetLen&**.

## Remarks

The server returns stored procedure information (including any return status and parameter values) immediately after returning all normal results for that stored procedure. Process the normal results, and then call **SqlRetData\$** after **SqlResults%** returns NOMORERPCRESULTS (for all stored procedures in a batch except the last one) or NOMORERESULTS (for a single stored procedure, or for the last stored procedure in a batch).

Stored procedures can return values for specified parameters. If the value of one of these parameters is changed in a stored procedure, the new value is returned to the program that called the procedure. This action parallels the pass-by-reference facility available in some programming languages.

To function as a return parameter, a parameter must be declared as such:

- When a stored procedure is created with CREATE PROCEDURE, the parameter must be declared as OUTPUT.
- For a stored procedure executed using the Transact-SQL EXECUTE statement, the parameter must be declared as OUTPUT.
- For a stored procedure executed using DB-Library functions (such as **SqlRpcInit%**), when the parameter is added using **SqlRpcParam%**, the *status%* bitmask must contain the SQLRPCRETURN option.

If a stored procedure is executed using DB-Library functions (such as **SqlRpcInit%**), the return parameter values are automatically available to the application. If a stored procedure is invoked with an EXECUTE statement, the return parameter values are available only if the command batch containing the EXECUTE statement uses Transact-SQL local variables, not constants, for the return parameters.

## See Also

[SqlNextRow%](#), [SqlNumRets%](#), [SqlResults%](#), [SqlRetLen&](#), [SqlRetName\\$](#), [SqlRetType%](#)

# SqlRetLen&

Determines the length of a return-parameter value generated by a stored procedure or a remote stored procedure.

## Syntax

**SqlRetLen&** ( *sqlconn%*, *retnum%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*retnum%*

Is the number of the returned value of interest. The first return value is number 1. Values are returned in the same order as the parameters were originally specified in the stored procedure's CREATE PROCEDURE statement. (Note that this is not necessarily the same order as that specified in the remote stored procedure.) When specifying *retnum%*, non-return parameters are not counted. For example, if the second parameter in a stored procedure is the only return parameter, its *retnum%* is 1, not 2.

## Returns

The length of the specified returned value. If *retnum%* is out of range, **SqlRetLen&** returns -1. If the return value is null, **SqlRetLen&** returns 0.

## Remarks

For more information about stored procedure return parameters, see "**SqlRetData\$**."

The server returns stored procedure information (including any return status and parameter values) immediately after returning all normal results for that stored procedure. Process the normal results, and then call **SqlRetLen&** after **SqlResults%** returns NOMORERPCRESULTS (for all stored procedures in a batch except the last one) or NOMORERESULTS (for a single stored procedure, or for the last stored procedure in a batch).

## See Also

[SqlNextRow%](#), [SqlNumRets%](#), [SqlResults%](#), [SqlRetData\\$](#), [SqlRetName\\$](#), [SqlRetType%](#)

# SqlRetName\$

Returns the name of a return parameter of a stored procedure or a remote stored procedure.

## Syntax

**SqlRetName\$** ( *sqlconn%*, *retnum%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*retnum%*

Is the number of the returned value of interest. The first return value is number 1. Values are returned in the same order as the parameters were originally specified in the stored procedure's CREATE PROCEDURE statement. (Note that this is not necessarily the same order as that specified in the remote stored procedure.) When specifying *retnum%*, non-return parameters are not counted. For example, if the second parameter in a stored procedure is the only return parameter, its *retnum%* is 1, not 2.

## Returns

The name of the parameter of the specified return value. If *retnum%* is out of range, **SqlRetName\$** returns an empty string.

## Remarks

For more information about stored procedure return parameters, see "**SqlRetData\$**."

The server returns stored procedure information (including any return status and parameter values) immediately after returning all normal results for that stored procedure. Process the normal results, and then call **SqlRetName\$** after **SqlResults%** returns NOMORERPCRESULTS (for all stored procedures in a batch except the last one) or NOMORERESULTS (for a single stored procedure, or for the last stored procedure in a batch).

## See Also

[SqlNextRow%](#), [SqlNumRets%](#), [SqlResults%](#), [SqlRetData\\$](#), [SqlRetLen&](#), [SqlRetType%](#)

# SqlRetStatus&

Returns the status number returned by a stored procedure or a remote stored procedure.

## Syntax

**SqlRetStatus& ( *sqlconn%* )**

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The return status number for the stored procedure or remote stored procedure.

## Remarks

Normally, completed stored procedures return a status number of 0. A RETURN statement in a stored procedure is used for application-specific return status numbers. If a RETURN statement is not used and an error occurs when the stored procedure is executed, SQL Server can return one of the following values:

Value	Description
-1	Missing object
-2	Datatype error
-3	Process was chosen as deadlock victim
-4	Permission error
-5	Syntax error
-6	Miscellaneous user error
-7	Resource error, such as out of space
-8	Nonfatal internal problem
-9	System limit was reached
-10	Fatal internal inconsistency
-11	Fatal internal inconsistency
-12	Table or index corrupt
-13	Database is corrupt
-14	Hardware error

Values -15 through -99 are reserved for future SQL Server expansion.

The **SqlHasRetStat%** function determines whether the most recently executed stored procedure generated a return status number.

The server returns stored procedure information (including any return status and parameter values) immediately after returning all normal results for that stored procedure. Process the normal results, and then call **SqlRetStatus&** after **SqlResults%** returns NOMORERPCRESULTS (for all stored procedures in a batch except the last one) or NOMORERESULTS (for a single stored procedure, or for the last stored procedure in a batch).

The order in which the application processes the status number and any returned parameter values is unimportant.

## See Also

SqlHasRetStat%, SqlNextRow%, SqlResults%, SqlRetData\$

# SqlRetType%

Determines the datatype of a return parameter value generated by a stored procedure or a remote stored procedure.

## Syntax

**SqlRetType%** ( *sqlconn%*, *retnum%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*retnum%*

Is the number of the returned value of interest. The first return value is number 1. Values are returned in the same order as the parameters were originally specified in the stored procedure's CREATE PROCEDURE statement. (Note that this is not necessarily the same order as that specified in the remote stored procedure.) When specifying *retnum%*, non-return parameters are not counted. For example, if the second parameter in a stored procedure is the only return parameter, its *retnum%* is 1, not 2.

## Returns

A token value for the datatype of the specified return value, as shown here:

Column datatype	Returned constant
<i>char</i>	SQLCHAR
<i>varchar</i>	SQLCHAR
<i>binary</i>	SQLBINARY
<i>varbinary</i>	SQLBINARY
<i>tinyint</i>	SQLINT1
<i>smallint</i>	SQLINT2
<i>int</i>	SQLINT4
<i>real</i>	SQLFLT4
<i>float</i>	SQLFLT8
<i>smallmoney</i>	SQLMONEY4
<i>money</i>	SQLMONEY
<i>decimal</i>	SQLDECIMAL
<i>numeric</i>	SQLNUMERIC
<i>smalldatetime</i>	SQLDATETIME4
<i>datetime</i>	SQLDATETIME

If *retnum%* is out of range, -1 is returned.

## Remarks

For more information about stored procedure return parameters, see "**SqlRetData\$**."

The server returns stored procedure information (including any return status and parameter values) immediately after returning all normal results for that stored procedure. Process the normal results, and then call **SqlRetType%** after **SqlResults%** returns NOMORERPCRESULTS (for all stored procedures in a batch except the last one) or NOMORERESULTS (for a single stored procedure, or for the last



stored procedure in a batch).

The **SqlRetType%** function actually returns an integer token value for the datatype (SQLCHAR, SQLFLT8, and so on). To convert the token value into a readable token string, use **SqlPrType\$**. For a list of all token values and their equivalent token strings, see [SQLPrType\\$](#).

#### **See Also**

[SqlNextRow%](#), [SqlNumRets%](#), [SqlPrType\\$](#), [SqlResults%](#), [SqlRetData\\$](#), [SqlRetLen&](#), [SqlRetName\\$](#)

# SqlRpcExec%

Executes a single stored procedure, a single remote stored procedure, or a batch of stored procedures and/or remote stored procedures on SQL Server.

## Syntax

**SqlRpcExec%** ( *sqlconn%* )

where

*sqlconn%*

A SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

After initializing and setting up each stored procedure using **SqlRpcInit%** and **SqlRpcParam%**, call **SqlRpcExec%** to execute the stored procedure or batch of stored procedures on SQL Server. Then call **SqlOk%** before processing the stored procedure results. **SqlRpcExec%** can be faster than **SqlRpcSend%** on some networks. For more information about executing stored procedures using DB-Library functions, see "**SqlRpcInit%**."

**Note** This function is one of the five (**SqlNextRow%**, **SqlResults%**, **SqlRpcExec%**, **SqlExec%**, **SqlOk%**) that do not return control to the application until after the server sends the required response. The application can be blocked for a considerable time if the server is waiting for a lock or is processing a large sort. If this is unacceptable, use **SqlRpcSend%** and **SqlOk%**, or set the DB-Library timeout to regain control periodically.

## See Also

[SqlNextRow%](#), [SqlResults%](#), [SqlRetData\\$](#), [SqlRetStatus&](#), [SqlRpcInit%](#), [SqlRpcParam%](#)

# SqlRpcInit%

Initializes a stored procedure or a remote stored procedure.

## Syntax

**SqlRpcInit%** ( *sqlconn%*, *rpcname\$*, *options%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*rpcname\$*

Is the name of the stored procedure to be invoked.

*options%*

Is a 2-byte bitmask of options for stored procedures. Specify 0 to indicate no options. The following options are available:

option	Description
SQLRPCRECOMPILE	Recompiles a stored procedure before it is executed.
SQLRPCRESET	Cancels a single stored procedure or a batch of stored procedures. If <i>rpcname\$</i> is specified, that new stored procedure is initialized after the cancel is complete.

## Returns

SUCCEED or FAIL.

## Remarks

An application can execute a single stored procedure, or it can execute a batch containing multiple stored procedures. To execute a single stored procedure, you can use DB-Library functions (such as **SqlRpcInit%**) or the Transact-SQL EXECUTE statement. To retrieve the status number and parameter values returned by each stored procedure in a batch, you must use DB-Library functions.

### ▶ To execute a single stored procedure or a batch of stored procedures using DB-Library functions:

1. Call **SqlRpcInit%** once to initialize a new stored procedure.
2. Call **SqlRpcParam%** for each parameter of the stored procedure that does not have a default value.
3. Repeat steps 1 and 2 for each stored procedure in the batch.
4. Call **SqlRpcSend%** or **SqlRpcExec%** to send the entire stored procedure batch to SQL Server.
5. Call **SqlOk%** to wait for SQL Server to start returning results.
6. Call **SqlResults%** to process the results from each stored procedure.  
If **SqlResults%** returns SUCCEED, call **SqlNextRow%** until it returns NOMOREROWS to process the normal results from the stored procedure.  
If **SqlResults%** returns NOMORERPCRESULTS, and you want to retrieve status number and return parameter information returned by the stored procedure, follow the steps given below.
7. Repeat step 6 until **SqlResults%** returns NOMORERESULTS.
8. If you want to retrieve status number and return parameter information returned by the last stored procedure in the batch, follow the steps given below.

After **SqlResults%** returns NOMORERPCRESULTS (for all stored procedures in a batch except the

last one) or NOMORERESULTS (for a single stored procedure, or for the last stored procedure in a batch), you can retrieve status number and return parameter information for a stored procedure.

► **To retrieve status number and return parameter information returned by a stored procedure using DB-Library functions:**

- Call **SqlRetStatus&** and **SqlHasRetStat%** to retrieve the return status number.
- Call **SqlNumRets%** to determine the number of return parameters.
- Call **SqlRetData\$**, **SqlRetType%**, **SqlRetLen&**, and **SqlRetName\$** about retrieve information for each return parameter.

Executing stored procedures with DB-Library functions has some advantages over using an EXECUTE statement:

- Using DB-Library functions to call a stored procedure causes DB-Library for Visual Basic to pass parameters in their native datatypes; using an EXECUTE statement passes them as ASCII characters. Calling stored procedures with DB-Library functions works faster and usually more efficiently than an EXECUTE statement because the server is not required to convert native datatypes into their ASCII equivalents.
- Using DB-Library functions instead of an EXECUTE statement accommodates return parameters for stored procedures more quickly. With a remote stored procedure, the return parameters are always available to the application by calling **SqlRetData\$**. (Note, however, that a return parameter must be specified as such when it is first added to the stored procedure through **SqlRpcParam%**.)

When a stored procedure is called with an EXECUTE statement, the return parameter values are available only if the command batch containing the EXECUTE statement uses local variables, rather than constants, as the return parameters. This involves additional parsing each time the command batch is executed.

- The client application can use DB-Library functions to issue a stored procedure call directly to an Open Data Services server application. The Open Data Services server application will detect this request as a remote stored procedure event. The Open Data Services server application is not required to parse the language buffer to find out what the client is requesting.

Stored procedures executed on the local SQL Server (using the *sqlconn%* connection) participate in transactions normally and can be rolled back. Remote stored procedures executed on a remote SQL Server cannot be rolled back.

**See Also**

[SqlNextRow%](#), [SqlResults%](#), [SqlRetData\\$](#), [SqlRetStatus&](#), [SqlRpcParam%](#), [SqlRpcSend%](#), [SqlOk%](#)

# SqlRpcParam%

Adds a parameter to a stored procedure or a remote stored procedure.

## Syntax

**SqlRpcParam%** ( *sqlconn%*, *paramname\$*, *status%*, *type%*, *maxlen&*,  
*datalen&*, *value\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*paramname\$*

Is the name of the parameter to be invoked. This name must begin with the at symbol (@), as do all parameter names within stored procedures. As in the Transact-SQL EXECUTE statement, the name is optional. If the name is not used, it should be specified as an empty string. In that case, the order of the **SqlRpcParam%** calls determines the parameter to which each call refers.

*status%*

Is a 1-byte bitmask of parameter options for stored procedures. The only option currently available is SQLRPCRETURN, which signifies that an application designates this parameter as a return parameter. Specify 0 to indicate no options.

*type%*

Is the datatype of the *value\$* parameter (such as SQLINT1, SQLCHAR, and so on).

*maxlen&*

For variable-length return parameters (when *type%* is SQLCHAR, SQLBINARY, SQLTEXT, or SQLIMAGE), the maximum desired byte length for the *value\$* parameter returned from a stored procedure.

Set *maxlen&* to -1 in any of these cases:

- For fixed-length return parameters (such as when *type%* is SQLINT4).
- To pass a NULL fixed-length parameter value (such as when *type%* is SQLINT4) to the stored procedure.
- For parameters that are not designated as return parameters.

Set *maxlen* to 0 to pass a NULL variable-length parameter value (when *type%* is SQLCHAR, SQLBINARY, SQLTEXT, or SQLIMAGE) to the stored procedure.

*datalen&*

For variable-length return parameters (when *type%* is SQLCHAR, SQLBINARY, SQLTEXT, or SQLIMAGE), *datalen&* is the actual byte length of the *value\$* parameter sent to the stored procedure. The byte length should *not* count any null terminator.

Set *datalen&* to - 1 for fixed-length parameters (such as when *type%* is SQLINT4).

Set *datalen&* to 0 to pass a NULL parameter value (fixed or variable length) to the stored procedure.

*value\$*

Is a string containing the stored procedure parameter itself. DB-Library converts the string to its native datatype. You must use binary strings for binary parameters (when *type%* is SQLBINARY or SQLIMAGE.)

The following table summarizes the required *maxlen&* and *datalen&* values for each type of parameter.

Parameter	<i>maxlen&amp;</i>	<i>datalen&amp;</i>
Fixed-length	- 1	- 1
Variable-length	Maximum desired	Length of input value

	length of return value	
Fixed-length NULL	- 1	0
Variable-length NULL	0	0

When specifying a NULL parameter, the actual contents of *value\$* is not used, and a NULL parameter is added to the stored procedure.

**Returns**

SUCCEED or FAIL.

**Remarks**

After initializing a stored procedure using **SqlRpcInit%**, you must call **SqlRpcParam%** once for each parameter of the stored procedure that does not have a default value. You specify default values for stored procedure parameters in the CREATE PROCEDURE statement. For more information about executing stored procedures using DB-Library functions, see "**SqlRpcInit%**."

**See Also**

[SqlNextRow%](#), [SqlOk%](#), [SqlResults%](#), [SqlRetData\\$](#), [SqlRetStatus&](#), [SqlRpcInit%](#), [SqlRpcSend%](#)

# SqlRpcSend%

Sends a single stored procedure, a single remote stored procedure, or batch of stored procedures and/or remote stored procedures to SQL Server to be executed.

## Syntax

**SqlRpcSend%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

SUCCEED or FAIL.

## Remarks

After initializing and setting up each stored procedure using **SqlRpcInit%** and **SqlRpcParam%**, call **SqlRpcSend%** to send the stored procedure or batch of stored procedures to SQL Server. Then call **SqlOk%** before processing the stored procedure results. For more information about executing stored procedures using DB-Library functions, see "**SqlRpcInit%**."

## See Also

[SqlNextRow%](#), [SqlOk%](#), [SqlResults%](#), [SqlRetData\\$](#), [SqlRetStatus&](#), [SqlRpcInit%](#), [SqlRpcParam%](#)

# Text and Image Functions

[SqlMoreText%](#)

[SqlTxPtr\\$](#)

[SqlTxTimeStamp\\$](#)

[SqlTxTsNewVal\\$](#)

[SqlTxTsPut%](#)

[SqlUpdateText%](#)

[SqlWriteText%](#)

[SqlTextUpdate1Row%](#)

[SqlTextUpdateManyRows&](#)



# SqlMoreText%

Sends part of a large text or image value to SQL Server.

## Syntax

**SqlMoreText%** ( *sqlconn%*, *size&*, *text\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*size&*

Is the size, in bytes, of the particular part of the text or image value sent to SQL Server. You cannot send more text or image bytes to SQL Server than are specified in the call to **SqlWriteText%** or **SqlUpdateText%**. The *size&* parameter cannot be larger than 32K.

*text\$*

Is a string containing the text or image portion to be written.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlMoreText%** is used in conjunction with **SqlWriteText%** or **SqlUpdateText%** to send a large text or image value to SQL Server in the form of a number of smaller chunks. This function is particularly useful with operating systems unable to allocate extremely long data buffers.

After calling **SqlMoreText%** for the last time, call **SqlOk%**.

For an example using **SqlMoreText%**, see "**SqlWriteText%**."

## See Also

[SqlTxPtr\\$](#), [SqlTxTimeStamp\\$](#), [SqlUpdateText%](#), [SqlWriteText%](#)

# SqlTxPtr\$

Return the identifier for a text or image column in the current row.

## Syntax

**SqlTxPtr\$** ( *sqlconn%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of the column. The first column in a table is number 1.

## Returns

The identifier for a text or image column in the current row. In the case of a NULL text or image value, the identifier value is an empty string.

**Important** Do not modify this identifier in any way. Modifying the identifier can cause unpredictable results.

## Remarks

Every text or image column has an associated identifier that uniquely identifies the text or image value. This identifier is useful in conjunction with **SqlWriteText%**. The identifier returned by **SqlTxPtr\$** supplies the value for the *textptr\$* parameter of **SqlWriteText%**. Call **SqlTxPtr\$** only after **SqlNextRow%** or **SqlGetRow%** has returned regrow.

Text pointers are of fixed length and can be null when the text or image value is null.

## Example

See the Windows-based programming example for **SqlWriteText%**, later in this chapter, which uses **SqlTxPtr\$**.

## See Also

[SqlTxTimeStamp\\$](#) and [SqlWriteText%](#)

# SqlTxTimeStamp\$

Returns the identifier for the text timestamp for a column in the current row.

## Syntax

**SqlTxTimeStamp\$** ( *sqlconn%*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of the column. The first column in a table is number 1.

## Returns

The identifier of the text timestamp for the column. This identifier can be an empty string.

**Important** Do not modify this identifier in any way. Modifying the identifier can cause unpredictable results.

## Remarks

Every text or image column has an associated text timestamp that marks the time of the column's last modification. The text timestamp is useful in conjunction with **SqlWriteText%** to ensure that two competing users do not inadvertently wipe out each other's modifications in the database. The text timestamp is returned to the *sqlconn%* when a Transact-SQL SELECT statement is performed on a SQLTEXT or SQLIMAGE column.

The length of a non-NULL text timestamp is always SQLTXTSLEN (currently defined as eight bytes). Call **SqlTxTimeStamp\$** only after **SqlNextRow%** or **SqlGetRow%** has returned REGROW.

## Example

See the Windows-based programming example for **SqlWriteText%**, later in this chapter, which uses **SqlTxTimeStamp\$**.

## See Also

[SqlTxPtr\\$](#) and [SqlWriteText%](#)

# SqlTxTsNewVal\$

Returns the identifier for the new value for a text timestamp after a call to **SqlWriteText%**.

## Syntax

**SqlTxTsNewVal\$** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The identifier for the timestamp value for the text or image value modified by a **SqlWriteText%** operation. This identifier can be an empty string.

**Important** Do not modify this identifier in any way. Modifying the identifier can cause unpredictable results.

## Remarks

Every text or image column has an associated text timestamp that is updated whenever the column's value is changed. The new text timestamp identifier, returned by **SqlTxTsNewVal\$**, can be used in conjunction with **SqlWriteText%** to ensure that two competing users do not inadvertently wipe out each other's modifications in the database. It is returned to the *sqlconn%* when a Transact-SQL SELECT statement is performed on a SQLTEXT or SQLIMAGE column and can be examined by calling **SqlTxTimeStamp\$**.

After each successful **SqlWriteText%** operation (which can include a number of calls to **SqlMoreText%**), SQL Server sends the updated value of *timestamp\$* back to DB-Library for Visual Basic. The application can then get the new value of *timestamp\$* with **SqlTxTsNewVal\$** and use **SqlTxTsPut%** to put that new value into the row buffer for future access through **SqlTxTimeStamp\$**. This capability is particularly useful when the application does not need the new timestamp immediately because row buffering is turned on.

## See Also

[SqlMoreText%](#), [SqlTxTimeStamp\\$](#), [SqlTxTsPut%](#), [SqlWriteText%](#)

# SqlTxTsPut%

Places the identifier for the new value for a text timestamp into a column of the current row in the row buffer.

## Syntax

**SqlTxTsPut%** ( *sqlconn%*, *newtxts\$*, *column%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*newtxts\$* ( )

Is the new text timestamp value, returned by **SqlTxTsNewVal\$**.

*column%*

Is the number of the column to receive the new text timestamp. Column numbers start at 1.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

Every text or image column has an associated text timestamp that is updated whenever the column's value is changed. The text timestamp is useful in conjunction with **SqlWriteText%** to ensure that two competing users do not inadvertently wipe out each other's modifications in the database. It is returned to the *sqlconn%* when a Transact-SQL SELECT statement is performed on a SQLTEXT or SQLIMAGE column and can be examined by calling **SqlTxTimeStamp\$**. Call **SqlTxTsPut%** only after **SqlNextRow%** or **SqlGetRow%** has returned REGROW.

After each successful **SqlWriteText%** operation (which can include a number of calls to **SqlMoreText%**), SQL Server sends the updated text timestamp value back to DB-Library for Visual Basic. **SqlTxTsNewVal\$** enables the application to get this new timestamp value. The application can then use **SqlTxTsPut%** to place the new timestamp value in the row buffer for future access through **SqlTxTimeStamp\$**. This is particularly useful when the application does not need the new timestamp immediately because row buffering is turned on.

## See Also

[SqlMoreText%](#), [SqlTxTimeStamp\\$](#) and [SqlTxTimeStamp&](#), [SqlTxTsNewVal\\$](#), [SqlWriteText%](#)

# SqlUpdateText%

Updates an existing *text* or *image* value. Unlike **SqlWriteText%** which replaces an entire *text* or *image* value, **SqlUpdateText%** can change only a portion of a *text* or *image* value in place.

## Syntax

**SqlUpdateText%** ( *sqlconn%*, *dest\_object\$*, *dest\_textptr\$*, *dest\_timestamp\$*, *update\_type%*,  
*insert\_offset&*, *delete\_length&*, *src\_object\$*, *src\_size&*, *src\_text\$* )

where

*sqlconn%*

A SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*dest\_object\$*

Is the destination table and *text* or *image* column name (separated with a period) of the existing *text* or *image* value to be updated.

*dest\_textptr\$*

Is the text pointer of the existing *text* or *image* value to be updated. Call **SqlTxPtr\$** to get this value. This string cannot be empty.

*dest\_timestamp\$*

Is the timestamp of the existing *text* or *image* value to be updated. Call **SqlTxTimeStamp\$** to get this value. This string cannot be empty.

*update\_type%*

The type of update operation to perform.

Use UTTEXT% to insert new data from a program variable specified in this call to **SqlUpdateText%**. You must specify an empty *src\_object\$* string, a *src\_size&* equal to the size in bytes of the new data value being inserted, and a non-empty *src\_text\$* that contains the new data value being inserted.

Use UTMORETEXT% to insert new data from a program variable with later calls to **SqlMoreText%**. You must specify an empty *src\_object\$* string, a *src\_size&* equal to the total size in bytes of the new data value (that will be inserted by calls to **SqlMoreText%**), and an empty *src\_text\$* string. Then call **SqlMoreText%** to insert the new data value in chunks.

Use UTTEXTPTR% to insert new data from a *text* or *image* column of an existing table. You must specify a non-empty *src\_object\$* string that gives the table and column a *src\_size&* of 0, and a non-empty *src\_text\$* string that gives the text pointer (returned by **SqlTxPtr\$**) of the new data value being inserted from the *src\_object\$*.

Use UTDELETEONLY% to only delete existing data. You must specify an *insert\_offset&* value other than -1 and a non-zero *delete\_length&* value. Specify that no new data will be inserted by using an empty *src\_object\$* string, a *src\_size&* of 0 and an empty *src\_text\$* string.

By default, a **SqlUpdateText%** operation is not recorded in the transaction log. You can bitwise OR the UTLOG% bit flag with any of these *update\_type%* values (for example UTTEXT% Or UTLOG%) to indicate that this operation will be recorded in the transaction log.

*insert\_offset&*

Is the zero-based starting position, specified as the number of bytes (from the start of the existing *text* or *image* value) to skip before inserting the new data. The existing *text* or *image* data beginning at this zero-based starting position will be shifted to the right to make room for the new data. A value of 0 means that the new data will be inserted at the beginning of the existing data value. A value of -1 means that the new data will be appended to the existing data value.

*delete\_length&*

Is the number of bytes to delete from the existing *text* or *image* value, starting at the *insert\_offset&* position. A value of 0 means that no data will be deleted. A value of -1 means that all data from the *insert\_offset&* position to the end of the existing *text* or *image* value will be deleted.

*src\_object\$*

Is the source table and *text* or *image* column name (separated with a period) that can be used as the source of the inserted data. If not empty, the *src\_size&* and *src\_text\$* parameters are ignored. If this string is empty, the *src\_size&* and *src\_text\$* parameters specify the data to be inserted.

*src\_size&*

Is the total size, in bytes, of the source *text* or *image* value (specified by *src\_text\$*) to be inserted.

Use this parameter only if the *src\_object\$* parameter is empty.

*src\_text\$*

Is a pointer to the source data value to be inserted. Use this parameter only if the *src\_object\$* parameter is empty.

## Returns

SUCCESS (1) or FAIL (0).

## Remarks

The **SqlUpdateText%** function can be used to delete existing data and then insert new data, to delete only existing data, or to insert only new data.

To delete existing data and then insert new data, specify an *update\_type%* other than UTDELETEONLY%, an *insert\_offset&* value other than -1, a non-zero *delete\_length&* value, and the new data to be inserted.

To delete only existing data, specify an *update\_type%* value of UTDELETEONLY%, an *insert\_offset&* value other than -1 and a non-zero *delete\_length&* value. Do not specify any new data to be inserted.

To insert only new data, specify an *update\_type%* other than UTDELETEONLY%, a *delete\_length&* value of 0, and the new data to be inserted.

## See Also

[SqlMoreText%](#), [SqlTxPtr\\$](#), [SqlTxTimeStamp\\$](#), [SqlTxTsNewVal\\$](#), [SqlTxTsPut%](#)

# SqlWriteText%

Sends a text or image value to SQL Server.

## Syntax

**SqlWriteText%** ( *sqlconn%*, *objname\$*, *textptr\$*, *textptrlen%*, *timestamp\$*,  
*log%*, *size&*, *text\$* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*objname\$*

Is the database table name and column name. The table name and the column name are separated by a period.

*textptr\$* ( )

Is the text or image value to be modified. This identifier can be obtained by calling **SqlTxPtr\$**.

*textptrlen%*

Is a parameter. This parameter is included for future compatibility. For now, its value must be the constant SQLTXPLEN.

*timestamp\$* ( )

Is the text timestamp for the text or image value to be modified. This identifier can be obtained by calling **SqlTxTimestamp\$**. The value changes whenever the text or image value itself is changed. This string cannot be empty.

*log%*

Is a Boolean value that specifies whether this **SqlWriteText%** operation should be recorded in the transaction log.

*size&*

Is the total size, in bytes, of the text or image value to be written.

*text\$*

Is a string containing the text or image to be written. If this string is empty, DB-Library for Visual Basic expects the application to call **SqlMoreText%** one or more times until all *size&* bytes of data have been sent to SQL Server. No single data block can be larger than 64K.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlWriteText%** is used to update text and image values, allowing an application to send long values to SQL Server without having to copy them into a Transact-SQL UPDATE statement. In addition, **SqlWriteText%** gives an application access to the text timestamp mechanism, which can be used to ensure that two competing users do not inadvertently wipe out each other's modifications in the database.

**SqlWriteText%** succeeds only if its *timestamp\$* parameter, usually obtained when the column's value is originally retrieved, matches the text column's timestamp in the database. If a match occurs, **SqlWriteText%** updates the text column and at the same time updates the column's timestamp. This has the effect of governing updates by competing applications – an application's **SqlWriteText%** call fails if a second application has updated the text column between the time the first application retrieved the column and the time it made its **SqlWriteText%** call.

**SqlWriteText%** is similar to a Transact-SQL WRITETEXT statement. However, calling **SqlWriteText%** is usually more efficient than sending a WRITETEXT statement through the command buffer. (For



information about WRITETEXT, see the *Microsoft SQL Server Transact-SQL Reference*.)

**SqlWriteText%** can be invoked with or without logging in, according to the value of the *log%* parameter. To use **SqlWriteText%** with logging turned off, the SQL Server option **select into/bulkcopy** must be set to TRUE by executing the following system procedure:

```
sp_dboption 'mssql', 'select into/bulkcopy', 'true'
```

For more information about SQL Server options, see the *Microsoft SQL Server Transact-SQL Reference* and the *Microsoft SQL Server Administrator's Companion*.

**SqlWriteText%**, used in conjunction with **SqlMoreText%**, also enables an application to send a large text or image value to SQL Server in the form of a number of smaller chunks. This is particularly useful with operating systems that are unable to allocate extremely long data buffers.

When **SqlWriteText%** is used with **SqlMoreText%**, it locks the specified database text column, and the lock is not released until the final **SqlMoreText%** has sent its data. This ensures that a second application does not read or update the text column in the middle of the first application's update.

If the *text\$* string is not an empty string, **SqlWriteText%** executes the data transfer from start to finish, including any necessary calls to **SqlOk%** and **SqlResults%**. To send a text or image value in chunks rather than sending the whole value at once, set the *text\$* parameter to an empty string.

**SqlWriteText%** returns control to the application immediately after notifying SQL Server that a text transfer is about to begin. The actual text is sent to SQL Server with **SqlMoreText%**, which can be called multiple times, once for each chunk.

## Examples

The following code fragment uses **SqlWriteText%**:

```
A.  'Retrieve a record from the "abstract" text column.
    cmd$ = "SELECT abstract FROM articles"
    cmd$ = cmd$ + " WHERE article_id = 1000"
    Result% = SqlCmd%(Sqlconn%, cmd$)
    Result% = SqlExec%(Sqlconn%)
    Result% = SqlResults%(Sqlconn%)
    Result% = SqlNextRow%(Sqlconn%)

    'Update the text column.
    Abstract$ = "A brand new text value."
    Result% = SqlWriteText%(UpdSqlconn%, "articles.abstract", _
        SqlTxPtr$(Sqlconn%, 1), SQLTXPLEN, _
        SqlTxTimestamp$(Sqlconn%, 1), 1, _
        LEN(Abstract$), Abstract$)
```

The following code fragment uses **SqlWriteText%** with **SqlMoreText%**. Notice the required calls to **SqlOk%** and **SqlResults%** between the call to **SqlWriteText%** and the first call to **SqlMoreText%** and after the final call to **SqlMoreText%**.

```
B.  'Retrieve a record from the "abstract" text column.
    cmd$ = "SELECT abstract FROM articles"
    cmd$ = cmd$ + " WHERE article_id = 1000"
    Result% = SqlCmd%(Sqlconn%, cmd$)
    Result% = SqlExec%(Sqlconn%)
    Result% = SqlResults%(Sqlconn%)

    Result% = SqlNextRow%(Sqlconn%)

    'Update the text column.
```

```

Abstr1$ = "A brand new text value."
Abstr2$ = " This text value contains two sentences."
Result% = SqlWriteText%(UpdSqlconn%, "articles.abstract", _
    SqlTxPtr$(Sqlconn%, 1), SQLTXPLEN, _
    SqlTxTimestamp$(Sqlconn%, 1), 1, _
    LEN(Abstr1$ + Abstr2$), "")
Result% = SqlOk%(UpdSqlconn%)
Result% = SqlResults%(UpdSqlconn%)

'Send the update value in chunks.
Result% = SqlMoreText%(UpdSqlconn%, LEN(Abstr1$), Abstr1$)
Result% = SqlMoreText%(UpdSqlconn%, LEN(Abstr2$), Abstr2$)
Result% = SqlOk%(UpdSqlconn%)
Result% = SqlResults%(UpdSqlconn%)

```

These examples specify datatypes for Windows.

### See Also

[SqlMoreText%](#), [SqlTxPtr\\$](#), [SqlTxTimeStamp\\$](#), [SqlTxTsNewVal\\$](#), [SqlTxTsPut%](#)

# SqlTextUpdate1Row%

Updates one row of results in a text column. All subsequent rows, if any, are removed from the results buffer.

## Syntax

**SqlTextUpdate1Row%** ( *sqlconn%*, *objname\$*, *column%*, *text\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*objname\$*

Is the database table and column name. The table name and the column name are separated by a period.

*column%*

Is the number of the column. The first column in a table is number 1.

*text\$*

Specifies a string containing the text or image replacing the current data on the server.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlTextUpdate1Row%** is a utility function that performs several actions to reduce the number of steps for updating the data in a text or image column of a single row. **SqlUpdate1Row%** is equivalent to calling **SqlTxPtr\$**, **SqlTxTsPut%**, and **SqlWriteText%**. The data is updated in the current row determined by a call to **SqlNextRow%**. You can call **SqlTextUpdate1Row%** any time after you call **SqlNextRow%**.

## See Also

[SqlTxPtr\\$](#), [SqlTxTsPut%](#), [SqlWriteText%](#), [SqlTextUpdateManyRows&](#)

# SqlTextUpdateManyRows&

Updates all rows of results in a text or image column, starting from the current row.

## Syntax

**SqlTextUpdateManyRows&** ( *sqlconn%*, *usqlconn%*, *objname\$*, *column%*, *text\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*usqlconn%*

Is a SQL Server connection used only for updating the desired column. The value of *usqlconn%* is returned by **SqlOpen%**.

*objname\$*

Is the database table and column name. The table name and the column name are separated by a period.

*column%*

Is the number of the column. The first column in a table is number 1.

*text\$*

Is a string containing the text or image replacing the current data on the server.

## Returns

The number of updated rows. If an error occurs, -1 is returned.

## Remarks

**SqlTextUpdateManyRows&** replaces the text or image of all rows in a column, starting from the current row. It is usually used to change the text or image of an entire column, starting from the first row.

**SqlTextUpdateManyRows&** performs several actions that reduce the number of steps for updating the data in a text or an image column of a single row. This function is equivalent to calling **SqlTxPtr\$**, **SqlTxTsPut%**, and **SqlWriteText%**. The data is updated in the current row, which is determined by a call to **SqlNextRow%**.

The *usqlconn%* connection must be clear — that is, it cannot have any rows pending. After completing all the updates to the rows, immediately close the *usqlconn%* connection.

Call **SqlTextUpdateManyRows&** any time after you call **SqlResults%**. Updates begin on the next row from the current pending row or, if **SqlNextRow%** has not been called, updates begin on the first row.

## See Also

[SqlTxPtr\\$](#), [SqlTxTsPut%](#), [SqlTextUpdate1Row%](#), [SqlWriteText%](#)

# Browse Functions

[SqlColBrowse%](#)

[SqlColSource\\$](#)

[SqlQual\\$](#)

[SqlTabBrowse%](#)

[SqlTabCount%](#)

[SqlTabName\\$](#)

[SqlTabSource\\$](#)

[SqlTsNewLen%](#)

[SqlTsNewVal\\$ and SqlTsNewVal&](#)

[SqlTsPut%](#)

[SqlTsUpdate%](#)

# SqlColBrowse%

Indicates whether the source of a result column can be updated with the DB-Library for Visual Basic browse-mode facilities.

## Syntax

**SqlColBrowse%** ( *sqlconn%*, *column%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of the result column. The first column is number 1.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlColBrowse%** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, see [DB-Library for Visual Basic Programming](#).

**SqlColBrowse%** determines whether the database column that is the source of a result column in a select list can be updated with the DB-Library for Visual Basic browse-mode facilities. **SqlColBrowse%** is useful for examining ad hoc queries. When a query has been hard-coded into the application, **SqlColBrowse%** is unnecessary.

You can call **SqlColBrowse%** any time after you call **SqlResults%**.

To determine the name of the source column, use **SqlColSource\$**.

## Example

Only a column derived from a table that has a unique index and a timestamp column can be updated. The column cannot be the result of a Transact-SQL expression. For example, in the following select list, result columns 1 and 2 (*title* and *category*) can be updated, but column 3 (*wholesale*) cannot because it is the result of an expression:

```
SELECT title, category=type, wholesale=(price * 0.6)
FROM inventory FOR BROWSE
```

## See Also

[SqlColSource\\$](#), [SqlQual\\$](#), [SqlTabBrowse%](#), [SqlTabCount%](#), [SqlTabName\\$](#), [SqlTabSource\\$](#), [SqlTsNewLen%](#), [SqlTsNewVal\\$](#), [SqlTsPut%](#)

# SqlColSource\$

Returns the name of the database column from which a result column derives.

## Syntax

**SqlColSource\$** ( *sqlconn%*, *tabnum%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*tabnum%*

Is the number of the table. The first table is number 1.

## Returns

A string containing the name of the database column from which a specified result derives.

**SqlColSource\$** returns an empty string if the column number is out of range or if the column is the result of a Transact-SQL expression, such as MAX(*colname*).

## Remarks

**SqlColSource\$** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, see [DB-Library for Visual Basic Programming](#). **SqlColSource\$** is useful for ad hoc queries. When a query has been hard-coded into the application, **SqlColSource\$** is unnecessary.

**SqlColSource\$** returns the underlying database column name. Don't confuse this column name with the optional column headings you can specify with a SELECT statement. For example, the following statement specifies a column header of "author" for the *au\_lname* column:

```
SELECT author = au_lname FROM authors
```

When updating a table, use the database column name, not the header name. The following code fragment uses **SqlColSource\$** to get the underlying database column name:

```
Source$ = SqlColSource$(Sqlconn%, 1)
```

You can call **SqlColSource\$** any time after you call **SqlResults%**.

## See Also

[SqlColBrowse%](#), [SqlQual\\$](#), [SqlTabBrowse%](#), [SqlTabCount%](#), [SqlTabName\\$](#), [SqlTabSource\\$](#), [SqlTsNewLen%](#), [SqlTsNewVal\\$](#) and [SqlTsNewVal&](#), [SqlTsPut%](#)

# SqlQual\$

Returns a string containing the WHERE clause for the current row in a specified table. In a browsable table, this string can be used to update the current row.

## Syntax

**SqlQual\$** ( *sqlconn%*, *tabnum%*, *tablename\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*tabnum%*

Specifies an integer to receive the number of the table. Tables are numbered in the order they are listed in the SELECT statement's FROM clause. Table numbers start at 1. If *tabnum%* is -1, *tablename\$* is used to identify the table.

*tablename\$*

Identifies a string containing the name of a table specified in the SELECT statement's FROM clause. If *tablename\$* is an empty string, *tabnum%* is used to identify the table.

## Returns

A string containing the WHERE clause for the current row in a specified table. If the specified table cannot be browsed, **SqlQual\$** returns an empty string. A browsable table has a unique index and a *timestamp* column.

## Remarks

**SqlQual\$** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, see [DB-Library for Visual Basic Programming](#).

**SqlQual\$** provides a WHERE clause that can be used to update a single row in a browsable table. Columns from this row must have been previously retrieved through a browse-mode SELECT query.

The WHERE clause produced by **SqlQual\$** begins with the keyword WHERE and contains references to the row's unique index and *timestamp* column. You can simply append the WHERE clause to an UPDATE or DELETE statement; there is no need to examine it or manipulate it in any way.

The *timestamp* column indicates the time that a particular row was last updated. An update on a browsable table fails if the *timestamp* column in the WHERE clause that **SqlQual\$** generates is different from the *timestamp* column in the table. Such a condition, which generates SQL Server error message 532, indicates that another user updated the row since it was selected for browsing. Design your application to include the logic for handling an update failure.

**SqlQual\$** can construct WHERE clauses only for browsable tables. You can use **SqlTabBrowse%** to determine whether a table can be browsed. **SqlQual\$** is usually called after **SqlNextRow%**.

## Example

The following code fragment illustrates one approach to updating a table in a multiuser environment using browse mode. The application attempts to update a record specified by the user. If the update fails, the application assumes that someone else already updated the same row and changed its timestamp. To handle this situation, the application simply repeats the process, retrieving the changed row for the user to examine and edit and allowing the user to decide whether to overwrite the change. A complete application examines the messages from the server to determine why the update failed.

DO

```
'Retrieve the employee record from the database.
```



```

'Assuming that "empid" is a unique index, the query
'will return only one row.
cmd$ = "SELECT * FROM employees WHERE empid ="
cmd$ = cmd$ + STR$(EmployeeId%) + " FOR BROWSE"
Result% = SqlCmd$(Sqlconn%, cmd$)
Result% = SqlExec$(Sqlconn%)
Result% = SqlResults$(Sqlconn%)
Result% = SqlNextRow$(Sqlconn%)

'Assign returned data to variables.
Name$ = SqlData$(Sqlconn%, 2)
Salary# = VAL(SqlData$(Sqlconn%, 3))

'Let user edit data.
PRINT "Name: "; Name$
PRINT "Salary: "; Salary#
INPUT "New salary"; Salary#

'Get a WHERE clause and update the database. Sqlconn%
'is the connection used to query the database, while
'UpdSqlconn% is the connection used to update it.
Qual$ = SqlQual$(Sqlconn%, -1, "employees")
Result% = SqlNextRow$(Sqlconn%)
cmd$ = "UPDATE employees SET salary ="
cmd$ = cmd$ + STR$(Salary#) + " " + Qual$
Result% = SqlCmd$(UpdSqlconn%, cmd$)
Result% = SqlExec$(UpdSqlconn%)

'If the update failed, try again.
LOOP WHILE SqlResults$(UpdSqlconn%) = FAIL
OR Result% = FAIL

```

### Output:

```

Name: Charles Dickens
Salary: 76543.21
New salary? 80000.00

```

### See Also

[SqlColBrowse%](#), [SqlColSource\\$](#), [SqlTabBrowse%](#), [SqlTabCount%](#), [SqlTabName\\$](#),  
[SqlTabSource\\$](#), [SqlTsNewLen%](#), [SqlTsNewVal\\$](#), [SqlTsPut%](#)

# SqlTabBrowse%

Indicates whether a specified table can be updated with the DB-Library for Visual Basic browse-mode procedures.

## Syntax

**SqlTabBrowse%** ( *sqlconn%*, *tabnum%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*tabnum%*

Is the number of the table as specified in the SELECT statement's FROM clause. Table numbers start at 1.

## Returns

SUCCEED (1) or FAIL (0). If you drop a table's unique index while browsing, **SqlTabBrowse%** continues to return SUCCEED.

## Remarks

**SqlTabBrowse%** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, see [DB-Library for Visual Basic Programming](#).

**SqlTabBrowse%** provides a way to identify browsable tables. A browsable table has a unique index and a timestamp column. **SqlColBrowse%** is useful when examining ad hoc queries prior to performing browse-mode updates based on them. When a query is hard-coded into the application, **SqlTabBrowse%** is unnecessary.

You can call **SqlTabBrowse%** any time after you call **SqlResults%**.

## See Also

[SqlColBrowse%](#), [SqlColSource\\$](#), [SqlQual\\$](#), [SqlTabCount%](#), [SqlTabName\\$](#), [SqlTabSource\\$](#), [SqlTsNewLen%](#), [SqlTsNewVal\\$](#), [SqlTsPut%](#)

# SqlTabCount%

Returns the number of tables included in the current SELECT statement.

## Syntax

**SqlTabCount%** ( *sqlconn%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The number of tables, including SQL Server work tables, included in the current SELECT statement. If an invalid *sqlconn%* value is sent to **SqlTabCount%**, a value of -1 is returned.

## Remarks

**SqlTabCount%** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, see [DB-Library for Visual Basic Programming](#).

A SELECT statement can generate a set of result rows whose columns are derived from several database tables. To perform browse-mode updates of the columns in a statement's select list, your application must know how many tables are involved in the query, because each table requires a separate UPDATE statement. **SqlTabCount%** can provide this information for ad hoc queries. When a query is hard-coded into the application, **SqlTabCount%** is unnecessary.

The count returned by **SqlTabCount%** includes any SQL Server work tables used in processing a query. SQL Server sometimes creates temporary internal work tables to process a query and deletes them by the time it finishes processing the statement. Work tables cannot be updated and are not available to your application. Therefore, design your application such that before it uses a table number, it verifies that the number does not belong to a work table. **SqlTabName\$** can be used to determine whether a particular table number refers to a work table.

You can call **SqlTabCount%** any time after you call **SqlResults%**.

## See Also

[SqlColBrowse%](#), [SqlColSource\\$](#), [SqlQual\\$](#), [SqlTabBrowse%](#), [SqlTabName\\$](#), [SqlTabSource\\$](#), [SqlTsNewLen%](#), [SqlTsNewVal\\$](#), [SqlTsPut%](#)

# SqlTabName\$

Returns the name of a table based on its number.

## Syntax

**SqlTabName\$** ( *sqlconn%*, *tabnum%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*tabnum%*

Is the number of a table. Table numbers start with 1. Use **SqlTabCount%** to find out the total number of tables involved in a particular statement.

## Returns

A string containing the name of a specified table. This string is empty if the table number is out of range or if the specified table is a SQL Server work table. For a description of work tables, see

[\*\*SqlTabCount%\*\*](#).

## Remarks

**SqlTabName\$** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, see [DB-Library for Visual Basic Programming](#).

A SELECT statement can generate a set of result rows whose columns are derived from several database tables. The database tables are specified by the FROM clause. **SqlTabName\$** provides a way for an application to determine the name of each table involved in an ad hoc query. When the query has been hard-coded into the application, **SqlTabName\$** is unnecessary.

You can call **SqlTabName\$** any time after you call **SqlResults%**.

## See Also

[\*\*SqlColBrowse%\*\*](#), [\*\*SqlColSource\\$\*\*](#), [\*\*SqlQual\\$\*\*](#), [\*\*SqlTabBrowse%\*\*](#), [\*\*SqlTabCount%\*\*](#), [\*\*SqlTabSource\\$\*\*](#), [\*\*SqlTsNewLen%\*\*](#), [\*\*SqlTsNewVal\\$\*\*](#), [\*\*SqlTsPut%\*\*](#)

# SqlTabSource\$

Returns the name and number of the table from which a result column is derived.

## Syntax

**SqlTabSource\$** ( *sqlconn%*, *column%*, *tabnum%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*column%*

Is the number of the result column. Column numbers start at 1.

*tabnum%*

Is an integer variable to receive the table number. Many DB-Library for Visual Basic functions that operate in browse mode accept either a table name or a table number. If **SqlTabSource\$** returns an empty string, *tabnum%* is set to -1.

## Returns

A string containing the name of the table from which a result column derives. If an empty string is returned, it means one of the following:

- The SQL Server connection is inactive. This is an error that invokes an application's error handler.
- The SELECT statement does not contain the FOR BROWSE clause.
- The column number is not in range.
- The column is the result of an expression, such as MAX(*colname*).

## Remarks

**SqlTabSource\$** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, see [DB-Library for Visual Basic Programming](#).

Use **SqlTabSource\$** to determine which tables provide the columns in the current set of result rows. This information is valuable when using **SqlQual\$** to construct WHERE clauses for UPDATE and DELETE statements based on ad hoc queries. When the query is hard-coded into the application, **SqlTabSource\$** is unnecessary. You can call **SqlTabSource\$** any time after you call **SqlResults%**.

## See Also

[SqlColBrowse%](#), [SqlColSource\\$](#), [SqlQual\\$](#), [SqlTabBrowse%](#), [SqlTabCount%](#), [SqlTabName\\$](#), [SqlTsNewLen%](#), [SqlTsNewVal\\$](#), [SqlTsPut%](#)

# SqlTsNewLen%

Returns the length of the new value of a timestamp column after a browse-mode update.

## Syntax

**SqlTsNewLen%** ( *sqlconn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The length, in bytes, of the updated row's new timestamp value. **SqlTsNewLen%** returns -1 if no timestamp is returned to the application because the update was unsuccessful or because the UPDATE statement did not contain a WHERE clause returned by **SqlQual\$**.

## Remarks

**SqlTsNewLen%** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, See [DB-Library for Visual Basic Programming](#).

**SqlTsNewLen%** provides information about the *timestamp* column. The WHERE clause returned by **SqlQual\$** contains references to the row's unique index and *timestamp* column. When you use such a WHERE clause in an UPDATE statement, a new value is placed in the updated row's *timestamp* column and a new timestamp value is returned to the application (if the update is successful). With **SqlTsNewLen%**, the application saves the length of the new timestamp value, possibly for use with **SqlTsPut%**.

## See Also

[SqlColBrowse%](#), [SqlColSource\\$](#), [SqlQual\\$](#), [SqlTabBrowse%](#), [SqlTabCount%](#), [SqlTabName\\$](#), [SqlTabSource\\$](#), [SqlTsNewVal\\$](#), [SqlTsPut%](#)

# SqlTsNewVal\$ and SqlTsNewVal&

Returns the identifier of the new value of a *timestamp* column after a browse-mode update.

## Syntax

**SqlTsNewVal\$** ( *sqlconn%* )

where

*sqlconn%* ( )

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

## Returns

The identifier of the updated row's new timestamp value. The identifier is an empty string if no timestamp is returned to the application because the update was unsuccessful or because the UPDATE statement did not contain a WHERE clause returned by **SqlQual\$**.

**Important** Do not modify the identifier in any way. Modifying the identifier can cause unpredictable results.

## Remarks

**SqlTsNewVal\$** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, see [DB-Library for Visual Basic Programming](#).

**SqlTsNewVal\$** provides information about the timestamp column. When used in an UPDATE statement, the WHERE clause returned by **SqlQual\$** places a new value in the updated row's timestamp column and returns the new timestamp value to the application (if the update is successful). With **SqlTsNewVal\$**, the application saves the new timestamp value, possibly for use with **SqlTsPut%**.

## See Also

[SqlColBrowse%](#), [SqlColSource\\$](#), [SqlQual\\$](#), [SqlTabBrowse%](#), [SqlTabCount%](#), [SqlTabName\\$](#), [SqlTabSource\\$](#), [SqlTsNewLen%](#), [SqlTsPut%](#)

# SqlTsPut%

Puts the new value of the timestamp column into a specified table's current row in the row buffer.

## Syntax

**SqlTsPut%** ( *sqlconn%*, *newts\$*, *newtslen%*, *tabnum%*, *tabname\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*newts\$*

Is the new timestamp value. The new timestamp value is returned by **SqlTsNewVal\$**.

*newtslen%*

Is the length of the new timestamp value. It is returned by **SqlTsNewLen%**.

*tabnum%*

Is the number of the table to receive the new timestamp. Table numbers start at 1. The *tabnum%* parameter must refer to a browsable table. Use **SqlTabBrowse%** to determine whether the table you specify can be browsed. If the table is browsable, *tabname\$* is used to identify the table.

*tabname\$*

Is a string containing the table name. The *tabname\$* parameter must refer to a browsable table. If the string is empty, *tabnum\$* is used to identify the table. The value of *tabname\$* is returned by **SqlTabSource\$**.

## Returns

SUCCEED (1) or FAIL (0). The following conditions cause **SqlTsPut%** to return FAIL:

- The application tries to update the timestamp of a nonexistent row.
- The application tries to update the timestamp using an empty string as the new timestamp identifier (*newts\$* or *newts&*).
- The specified table cannot be browsed.

## Remarks

**SqlTsPut%** is a DB-Library for Visual Basic browse-mode function. For a detailed discussion of browse mode, see [DB-Library for Visual Basic Programming](#).

**SqlTsPut%** manipulates the timestamp column. When used in an UPDATE statement, the WHERE clause returned by **SqlQual\$** places a new value in the updated row's timestamp column and returns the new timestamp value to the application (if the update is successful). If the same row is updated a second time, the UPDATE statement's WHERE clause must use the latest timestamp value.

**SqlTsPut%** updates the timestamp in the row currently being browsed. Then, if the application has to update the row a second time, it calls **SqlQual\$** to formulate a new WHERE clause that uses the new timestamp. With **SqlTsNewVal\$**, the application saves a new timestamp value, possibly for use with **SqlTsPut%**. Call **SqlTsPut%** only after **SqlNextRow%** or **SqlGetRow%** has returned regrow.

## See Also

[SqlColBrowse%](#), [SqlColSource\\$](#), [SqlQual\\$](#), [SqlTabBrowse%](#), [SqlTabCount%](#), [SqlTabName\\$](#), [SqlTabSource\\$](#), [SqlTsNewLen%](#), [SqlTsNewVal\\$](#)



# SqlTsUpdate%

Updates the value of a timestamp column in a specified table.

## Syntax

**SqlTsUpdate%** ( *sqlconn%*, *usqlconn%*, *tabnum%*, *tabname\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*usqlconn%*

Is a SQL Server connection used only for updating the timestamp of the desired column. The value of *usqlconn%* is returned by **SqlOpen%**.

*tabnum%*

Is the number of the table to receive the new timestamp. Table numbers start at 1. The *tabnum%* parameter must refer to a browsable table. Use **SqlTabBrowse%** to determine whether the table can be browsed. If this value is 1, the *tabname\$* parameter is used to identify the table.

*tabname\$*

Is a string containing the table name. The *tabname\$* parameter must refer to a browsable table. If the string is empty, *tabnum%* is used to identify the table. The value of *tabname\$* is returned by **SqlTabSource\$**.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlTsUpdate%** is equivalent to calling **SqlTsNewVal\$** and **SqlTsPut%**. Like those functions, it is designed for use in browse mode. **SqlTsUpdate%** can update the timestamp column if either *tabnum%* or *tabname\$* is provided. If neither is provided, calling **SqlTsUpdate%** results in an error.

The *usqlconn%* connection must be clear — that is, it cannot have any rows pending. After completing all the updates to the timestamp column, immediately close the *usqlconn%* connection.

## See Also

[SqlTabBrowse%](#), [SqlTabSource\\$](#), [SqlTsNewVal\\$](#), [SqlTsPut%](#)

# Bulk-Copy Functions

[SqlBCPColfmt%](#)

[SqlBCPColumns%](#)

[SqlBCPControl%](#)

[SqlBCPExec%](#)

[SqlBCPInit%](#)

[SqlBCPSetL%](#)

[SqlBCPColumnFormat%](#)

# SqlBCPColfmt%

Specifies the format of an operating-system file in a bulk copy into or out of SQL Server.

## Syntax

**SqlBCPColfmt%** ( *sqlconn%*, *fcolumn%*, *ftype%*, *fplen%*, *flen&*, *fterm\$*, *flen%*, *tcoll%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*fcolumn%*

Is the column number in the operating-system file whose format is being specified. The first column is number 1.

*ftype%*

Is the datatype of this column in the operating-system file. If the specified datatype is different from the datatype of the corresponding column in the database table (*tcoll%*), the datatype is converted automatically. To specify the same datatype as in the corresponding column of the database table (*tcoll%*), set this parameter to 0.

For a bulk copy out of SQL Server into a file, when *ftype%* is SQLDECIMAL or SQLNUMERIC:

- If the source column is not *decimal* or *numeric*, the default precision and scale are used.
- If the source column is *decimal* or *numeric*, the precision and scale of the source column are used.

*fplen%*

Is the length of the length prefix for this column in the operating-system file. Legal prefix lengths are 1, 2, and 4 bytes. To avoid using a length prefix, set this parameter to 0. So that DB-Library will determine whether to use a length prefix, set this parameter to -1. In that case, DB-Library uses a length prefix of whatever length is necessary if the database column length is variable.

If more than one means of specifying the column length of an operating-system file is used (such as a length prefix and a maximum column length, or a length prefix and a terminator sequence), DB-Library uses the one that results in the shortest amount of data being copied.

One valuable use of length prefixes is to simplify the specification of null data values in an operating-system file. For example, assume that you have a 1-byte length prefix for a 4-byte integer column. Ordinarily, the length prefix contains a value of 4 to indicate that a 4-byte value follows. However, if the value of the column is NULL, the length prefix can be set to 0 to indicate that 0 bytes follow.

*flen&*

Is the maximum length of this column's data in the operating-system file, not including the length of any length prefix and/or terminator. Setting *flen&* to 0 signifies that the data is NULL. Setting *flen&* to -1 directs the system to ignore this parameter, indicating that there is no default maximum length.

For fixed-length datatypes such as integers, the length of the data is constant, except in the special case of null values. Therefore, for fixed-length datatypes, *flen&* must always be -1 except when the data is NULL, in which case *flen&* must be 0. For character, text, binary, and image data, *flen&* can be

- 1, 0, or any positive value. When *flen&* is -1, the system uses either a length prefix or a terminator sequence to determine the length of the data. (If both are supplied, the system uses the one that results in the shorter amount of data being copied.) If *flen&* is -1 and neither a prefix length nor a terminator sequence is specified, the system returns an error message. If *flen&* is 0, the system assumes the data is NULL. If *flen&* is a positive value, the system uses *flen&* as the maximum data length. However, if a positive *flen&* and a prefix length and/or terminator sequence are provided, the system determines the data length by using the method that results in the shortest amount of data being copied.

*fterm\$*

Is the terminator sequence to be used for this column. This parameter is useful primarily for character, text, binary, and image datatypes, because all other types are of fixed length. To avoid using a terminator, set this parameter to NULL. To set the terminator to NULL, set *fterm\$* to *chr\$(0)*. To make the tab character the terminator, set *fterm\$* to *chr\$(9)*. To make the newline character the terminator, set *fterm\$* to *chr\$(13) + chr\$(10)*.

If more than one means of specifying the column length of an operating-system file is used (such as a terminator and a length prefix, or a terminator and a maximum column length), DB-Library uses the one that results in the shortest amount of data being copied.

***frlen%***

Is the length, in bytes, of the terminator sequence to be used for this column. To avoid using a terminator, set this value to 0.

***tcol%***

Is the corresponding column number in the database table. If this value is 0, this column is not copied. The first column number is column 1.

**Returns**

SUCCEED (1) or FAIL (0).

**Remarks**

**SqlBCPColfmt%** specifies the format of the operating-system file for bulk copies. The format of a bulk-copy operation contains the following parts:

- A mapping from the columns in the operating-system file to columns in the database.
- The datatype of each column in the operating-system file.
- The length of the optional-length prefix of each column.
- The maximum length of the data in a column in the operating-system file.
- The optional terminating-byte sequence in each column.
- The length of the optional terminating-byte sequence in each column.

Each call to **SqlBCPColfmt%** specifies the format for one column in the operating system file. For example, when you have a table with five columns and want to change the default settings for three of those columns, first call **SqlBCPColumns%(5)**, and then call **SqlBCPColfmt%** five times, with three of those calls setting your custom format. Set *ftype%* for the remaining two calls to 0 and *fplen%*, *fcrlen%*, and *frlen%* to -1. These settings copy all five columns – three with your customized format and two with the default format.

You must call **SqlBCPColumns%** before you call **SqlBCPColfmt%**.

You must call **SqlBCPColfmt%** once and only once for every column in the operating-system file, regardless of whether those columns use the default format or are skipped.

To skip a column, set *tcol%* to 0.

**See Also**

[SqlBCPColumnFormat%](#), [SqlBCPColumns%](#), [SqlBCPControl%](#), [SqlBCPExec%](#), [SqlBCPInit%](#)

# SqlBCPColumns%

Sets the total number of columns in the operating-system file for a bulk copy into or out of SQL Server.

## Syntax

**SqlBCPColumns%** ( *sqlconn%*, *colcount%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*colcount%*

Is the total number of columns in the operating-system file. Note that even when you are bulk-copying data from the operating-system file to a SQL Server table and do not intend to copy all the columns in the operating-system file, you must still set *colcount%* to the total number of columns in the operating-system file.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlBCPColumns%** can be called only after you call **SqlBCPInit%** with a valid filename.

Call this function only when you intend to use a format for an operating-system file that differs from the default format. For a description of the default format for an operating-system file, see [SqlBCPInit%](#).

After calling **SqlBCPColumns%**, you must call **SqlBCPColfmt%** *colcount%* times because you are defining a custom file format.

## See Also

[SqlBCPColfmt%](#), [SqlBCPColumnFormat%](#), [SqlBCPInit%](#)

# SqlBCPControl%

Changes the default settings for various control parameters for a bulk copy between a file and SQL Server.

## Syntax

**SqlBCPControl%** ( *sqlconn%*, *param%*, *value&* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*param%*

One of the following:

**BCPMAXERRS%**

Specifies the number of errors allowed before terminating. The default is 10. Providing a value of less than 1 to this field resets it to its default value. If a value larger than 65,535 is specified, this field is set to 65,535.

**BCPFIRST%**

Specifies the first row to copy. The default is 1. Providing a value of less than 1 to this field resets it to its default value.

**BCPLAST%**

Specifies the last row to copy. The default is to copy all rows. Providing a value of less than 1 to this field resets it to its default value.

**BCPBATCH%**

Specifies the number of rows per batch. The default is 0. Providing a value of less than 1 to this field resets it to its default value.

**BCPKEEPNULLS**

Specifies whether empty data values in the file will be converted to NULL values in the SQL Server table. If this option is set before calling **SqlBCPExec%**, empty values will be converted to NULL values in the SQL Server table. The default is for empty values to be converted to the column's default value in the SQL Server table.

*value&*

Is the value for the specified *param%*.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlBCPControl%** sets various control parameters for bulk-copy operations, including the number of errors allowed before ending an operation, the numbers of the first and last rows to copy, and the batch size.

These control parameters are meaningful only when you are copying between an operating-system file and a SQL Server table.

## Example

```
'Initialize bcp.
Result% =
SqlBCPInit%(Sqlconn%,"condb..address", "address.add", "addr.err", DBIN%)

IF Result% = FAIL THEN
```

```

        SQLExit
        SQLWinExit
    End
END IF
'Set the number of rows per batch.
Result% = SqlBDPControl%(Sqlconn%, BCPBATCH%, 1000)
IF Result% = FAIL THEN
    PRINT "SqlBCPControl% failed to set batching behavior."
    PRINT
    SQLExit
    SQLWinExit
    End
END IF

'Set file column count.
Result% = SQLBCPColumns%(Sqlconn%, 1)
IF Result% = FAIL THEN
    PRINT "SqlBCPColumns% failed."
    PRINT
    SQLExit
    SQLWinExit
    End
END IF

'Set file format.
Result% = SqlBCPColfmt%(Sqlconn%, 1, 0, 0, -1, Chr$(13), 1, 1)
IF Result% = FAIL THEN
    PRINT "SqlBCPColfmt% failed."
    PRINT
    SQLExit
    SQLWinExit
    End
END IF

'Now, execute the bulk-copy.
Result% = SqlBCPExec%(Sqlconn%, RowsCopied&)
IF Result% = FAIL THEN
    PRINT "Incomplete bulk-copy. Only "; RowsCopied&; " rows copied."
    PRINT
    SQLExit
    SQLWinExit
    End
END IF

```

### See Also

[SqlBCPColfmt%](#), [SqlBCPColumns%](#), [SqlBCPExec%](#), [SqlBCPInit%](#)

# SqlBCPExec%

Executes a bulk-copy of data between a database table and an operating-system file.

## Syntax

**SqlBCPExec%** ( *sqlconn%*, *rowscopied&* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*rowscopied&*

Is the number of rows successfully copied.

## Returns

SUCCEED (1) or FAIL (0). The **SqlBCPExec%** function returns SUCCEED only when all rows are copied. If some or none of the rows are copied, **SqlBCPExec%** returns FAIL. Check the *rowscopied&* parameter for the number of rows successfully copied.

## Remarks

**SqlBCPExec%** copies data from an operating-system file to a database table or vice versa, depending on the value of the *direction%* parameter in **SqlBCPInit%**.

Before calling **SqlBCPExec%**, you must call **SqlBCPInit%** with a valid operating-system filename. Failure to do so results in an error.

## Example

```
'Initialize bcp.
Result% =
SqlBCPInit%(Sqlconn%,"pubs..authors", "authors.sav", "", DBOUT%)
IF Result% = FAIL THEN
    SQLExit
    SQLWinExit
End
END IF

'Now, execute the bulk-copy.
Result% =
SqlBCPExec%(Sqlconn%, RowsCopied&)
IF Result% = FAIL THEN
    PRINT "Incomplete bulk-copy. Only "; RowsCopied&; " rows copied."
    PRINT
    SQLExit
    SQLWinExit
End
END IF
```

## See Also

[SqlBCPColfmt%](#), [SqlBCPColumns%](#), [SqlBCPControl%](#), [SqlBCPInit%](#)



# SqlBCPInit%

Initializes a bulk-copy operation.

## Syntax

**SqlBCPInit%** ( *sqlconn%*, *tblname\$*, *hfile\$*, *errfile\$*, *direction%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*tblname\$*

Is the name of the database table to copy. This name can also include the name of the database and the name of the database owner. For example, *pubs.gracie.titles*, *pubs..titles*, *gracie.titles*, and *titles* are all legal table names.

If *direction%* is DBOUT%, *tblname\$* can also be the name of a database view.

*hfile\$*

Is the name of the operating-system file to bulk copy into or out of SQL Server.

*errfile\$*

Is the name of the error file to be used. This error file contains progress messages, error messages, and copies of any rows that could not be copied from an operating-system file to a SQL Server table. If NULL is passed as *errfile\$*, no error file is used.

*direction%*

Is the direction of the copy. This parameter must be one of two values: DBIN% or DBOUT%. DBIN% indicates a copy from an operating-system file into the database table; DBOUT% indicates a copy from a database table into an operating-system file.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlBCPInit%** performs the initializations necessary to bulk-copy data between a workstation and SQL Server. It sets the default data formats for the operating-system file and examines the structure of the database table.

When you use an operating-system file to bulk-copy (see the description of the *hfile\$* parameter), the default native data formats are as follows:

- The order, type, length, and number of the columns in the operating-system file are assumed to be identical to the order, type, length, and number of the columns in the database table.
- If the data in a given database column is of fixed length, then the data column in the operating-system file is also of fixed length. If the data in a given database column is of variable length or can contain NULL values, the data column in the operating-system file is prefixed by a 4-byte length value for SQLTEXT and SQLIMAGE datatypes and a 1-byte length value for all other types.
- There are no terminators between columns in the operating-system file.

You can override any of these defaults by calling **SqlBCPColumns%** and **SqlBCPColfmt%**.

To use the bulk-copy functions to copy data to a database table, you must do the following:

- Call **SqlBCPSetL%** to make the server connection usable for bulk-copy purposes:

```
login% = sqllogin%  
SqlBCPSetL%(login%, TRUE)
```

- If the table has no indexes, set the database option **select into/bulkcopy** to TRUE, use the database, and send the SQL Server CHECKPOINT statement:

```
cmd$ = "sp_dboption 'mydb', 'select into/bulkcopy', 'true'"
Result% = SqlExec%(Sqlconn%)
Result% = SqlUse%(Sqlconn%, Database$)
cmd$ = "CHECKPOINT"
Result% = SqlExec%(Sqlconn%)
```

The **SqlBCPInit%** function must be called before any other bulk-copy functions. Failure to do so results in an error.

#### **See Also**

[SqlBCPColfmt%](#), [SqlBCPColumns%](#), [SqlBCPControl%](#), [SqlBCPExec%](#), [SqlBCPSetL%](#)

# SqlBCPSetL%

Sets the *loginrec%* to enable bulk-copy operations.

## Syntax

**SqlBCPSetL%** ( *loginrec%*, *enable%* )

where

*loginrec%*

Is a login record that is passed as a parameter to **SqlOpen%**. You get a login record by calling **SqlLogin%**.

*enable%*

Is a Boolean value, TRUE ( - 1 or 1) or FALSE (0), that specifies whether or not to enable bulk-copy operations for the resulting SQL Server connection. By default, SQL Server connections are not enabled for bulk-copy operations.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlBCPSetL%** sets a field in the login record that tells SQL Server that the *sqlconn%* connection can be used for bulk-copy operations. For **SqlBCPSetL%** to be effective, you must call it before you call **SqlOpen%**. **SqlOpen%** is the function that actually allocates the SQL Server connection.

To keep users from initiating a bulk-copy sequence with SQL statements, avoid using this function in applications that permit ad hoc queries. Once a bulk-copy sequence begins, it cannot be stopped with an ordinary SQL statement.

## See Also

[SqlBCPInit%](#), [SqlLogin%](#), [SqlOpen%](#)

# SqlBCPColumnFormat%

Sets up the column format for the input file in a bulk-copy operation. This is a utility function that combines **SqlBCPColumns%** and **SqlBCPColfmt%** in one step.

## Syntax

**SqlBCPColumnFormat%** ( *sqlconn%*, *col( )*, *numcols%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*col( )*

Is an array of a user-defined datatype (structure) defined as *BCPColData*, that contains the following elements:

*ftype%*

Is the datatype of this column in the operating-system file. If the datatype of this column is different from the datatype of the corresponding column in the database table (*tcol%*), the data is converted automatically. To specify the same data type as in the corresponding column in the database table (*tcol%*), set this parameter to 0.

*fplen%*

Is the length of the length prefix for this column in the operating-system file. Legal prefix lengths are 1, 2, and 4 bytes. To avoid using a length prefix, set this parameter to 0. So that the **bcp** utility will determine whether to use a length prefix, set this parameter to -1. In that case, the **bcp** utility uses a length prefix of whatever length is necessary if the database column length is variable.

When more than one means of specifying the column length of an operating-system file is used (such as a length prefix and a maximum column length, or a length prefix and a terminator sequence), the **bcp** utility uses the one that results in the shortest amount of data being copied.

One valuable use for length prefixes is to simplify the specifying of null data values in an operating-system file. For example, assume that you have a length prefix of 1 byte for a 4-byte integer column. Ordinarily, the length prefix contains a value of 4 to indicate that a 4-byte value follows. However, if the value of the column is an empty string, the length prefix can be set to 0 to indicate that 0 bytes follow.

*fcollen&*

Is the maximum length of this column's data in the operating-system file, not including the length of any length prefix and/or terminator. Setting *fcollen&* to 0 signifies that the data is an empty string. Setting *fcollen&* to -1 directs the system to ignore this parameter, indicating that there is no default maximum length.

For fixed-length datatypes such as integers, the length of the data is constant, except for the special case of null values. Therefore, for fixed-length datatypes, *fcollen&* must always be -1 except when the data is an empty string, in which case *fcollen&* must be 0. For character, text, binary, and image data, *fcollen&* can be -1, 0, or any positive value. If *fcollen&* is -1, the system uses either a length prefix or a terminator sequence to determine the length of the data. (If both are supplied, the system uses the one that results in the shorter amount of data being copied.) If *fcollen&* is -1 and neither a prefix length nor a terminator sequence is specified, the system returns an error message. If *fcollen&* is 0, the system assumes that the data is an empty string. If *fcollen&* is a positive value, the system uses *fcollen&* as the maximum data length. However, if a positive *fcollen&* and a prefix length and/or terminator sequence are provided, the system determines the data length by using the method that results in the shortest amount of data being copied.

*fterm\$*

Is the terminator sequence to be used for this column. This parameter is useful for *character*, *text*,

*binary*, and *image* datatypes, because all other types are of fixed length. To avoid using a terminator, set this parameter to an empty string. To set the terminator to null-terminated, set *fterm\$* to *chr\$(0)*. To make the tab character the terminator, set *fterm\$* to *chr\$(9)*. To make the newline character the terminator, set *fterm\$* to *chr\$(13) + chr\$(10)*. The length of *fterm\$* is limited to 30 characters.

When more than one means of specifying the column length of an operating-system file is used (such as a terminator and a length prefix, or a terminator and a maximum column length), the **bcp** utility uses the one that results in the shortest amount of data being copied.

*ftlen%*

Is the length, in bytes, of the terminator sequence to be used for this column. To avoid using a terminator, set this value to -1.

*tcol%*

Is the corresponding column in the database table. If this value is 0, the column in the database table is not copied. The first column is column 1.

*numcols%*

Is the total number of columns to be copied.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlBCPColumnFormat%** is equivalent to calling **SqlBCPColumns%** (to set the number of columns) and repeated calls to **SqlBCPColfmt%** (to set the format).

The size of the *col( )* array is equal to the number of columns (*numcols%*). To access an element of the *col* structure, you must use the point (.) notation. For example:

```
col.elementname
```

For example, the following statement sets the datatype of the column in the operating-system file:

```
Col.ftype% = SystemFileType%
```

## See Also

[SqlBCPColfmt%](#), [SqlBCPColumns%](#)

# Utility Functions

[SqlGetAltCollInfo%](#)

[SqlGetColumnInfo%](#)

[SqlOpenConnection%](#)

[SqlSendCmd%](#)

# SqlGetAltCollInfo%

Gets the column ID, datatype, maximum length, type of aggregate function, and aggregate type name of a specific column in the current set of results.

## Syntax

**SqlGetAltCollInfo%** ( *sqlconn%*, *altcolumnndata*, *computeid%*, *altcolumn%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*altcolumnndata*

Is a user-defined datatype (structure) defined as *AltColumnData*. It contains the following elements:

*colid%*

Is an integer value of the column identification number that the aggregate of the compute column applies to.

*datatype%*

Is the datatype of the compute column. See **SqlAltType%** for the list of datatype token values. If either the *computeid%* or the *altcolumn%* is invalid, this value is - 1.

*maxlen&*

The maximum length, in bytes, that the data in the column can be.

*aggtype%*

The type of aggregate operator of the compute column. See **SqlAltOpt%** for the list of aggregate function types.

*aggopname\$*

A string containing the name of the aggregate operator used by SQL Server – for example, SUM, AVG, and MAX.

*computeid%*

Is the compute ID. A SELECT statement can have multiple COMPUTE clauses, each of which can have a different number of aggregate operators and aggregate targets. The *computeid%* is returned by **SqlNextRow%** or **SqlGetRow%**.

*altcolumn%*

Is the number of the column. The first column returned is number 1.

## Returns

SUCCEED (1) or FAIL (0). The column ID, datatype, maximum length, type of aggregate function, and name of the aggregate operator type are returned by the *altcolumnndata* parameter.

## Remarks

**SqlGetAltCollInfo%** performs the actions of several other Visual Basic functions and puts the results of the functions in the corresponding elements of the *altcolumnndata* parameter. **SqlGetAltCollInfo%** is the equivalent of calling **SqlAltCollId%**, **SqlAltLen%**, **SqlAltOp%**, and **SqlAltType%**. To access an element returned by *altcolumnndata*, you must use the point (.) notation:

```
altcolumnndata.elementname
```

For example, the following statement gets the column name returned by *altcolumnndata* :

```
AggregateOperatorName$ = AltColumnData.AggOpName$
```

## Example

```

Dim AltColDetails As AltColumnData
'Put commands into the command buffer.
cmd$ = "SELECT title_id, price, advance FROM titles"
cmd$ = cmd$ + " COMPUTE SUM(price), MAX(advance) "
Result% = SqlCmd%(SqlConn%, cmd$)
'Send the command to SQL Server and start execution.
Result% = SqlExec%(SqlConn%)
Result% = SqlResults%(SqlConn%)

'Get the column ID, datatype, and name of the aggregate operator from
compute columns and print the results.
IF Result% = SUCCEED THEN
    DO UNTIL SqlNextRow%(SqlConn%) = NOMOREROWS
        IF Result% = REGROW THEN
            PRINT "regular row returned."
            PRINT
        ELSE
            GetAltColInfo% = SqlGetAltColInfo%(SqlConn%, AltColDetails,
                ComputeId%, AltColumn%)
            IF GetAltColInfo% = SUCCEED THEN
                AltColumnID% = AltColDetails.ColID%
                ColDatatype% = AltColDetails.Datatype%
                AggregateOperator$ = AltColDetails.AggOpName$
                PRINT "Column ID of compute column = " + AltColumnID%
                PRINT "Datatype of column = " + ColData type%
                PRINT "Aggregate Operator of column = " + AggregateOperator$
                PRINT
            END IF
        END IF
    LOOP

```

### See Also

[SqlAltColId%](#), [SqlAltLen%](#), [SqlAltOp%](#), [SqlAltType%](#), [SqlGetColumnInfo%](#)



# SqlGetColumnInfo%

Gets the type, length, name, and server datatype of a specific column in the current set of results.

## Syntax

**SqlGetColumnInfo%** ( *sqlconn%*, *columndata*, *column%* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*columndata*

Is a user-defined datatype (structure), defined as *ColumnData*. It contains the following elements:

*coltype%*

Is an integer value for the datatype of a particular column. For the list of datatype token values, see "**SqlColType%**".

*collen&*

The maximum length of the column.

*colname\$*

A string containing the name of a particular column. The string is 30 characters long.

*colsqltype\$*

A string containing the name of the datatype used by SQL Server – for example, *int*, *smallint*, and *text*. The string is 30 characters long.

*column%*

Is the number of the column. The first column is number 1.

## Returns

SUCCEED (1) or FAIL (0). The column type, length, name, and SQL Server datatype are returned by the *columndata* parameter.

## Remarks

**SqlGetColumnInfo%** performs the actions of several other DB-Library for Visual Basic functions and puts the results of those functions in the corresponding elements of the *columndata* parameter.

**SqlGetColumnInfo%** is the equivalent of calling **SqlColType%**, **SqlColLen%**, and **SqlColName\$**. To access the information about an element returned by *columndata*, you must use the point (.) notation:

```
columndata.elementname
```

For example, the following statement gets the column name returned by *columndata*:

```
ColumnName$ = columndata.colname$
```

You can call **SqlGetColumnInfo%** any time after you call **SqlResults%**.

## Example

```
'Put command into the command buffer.
cmd$ = "SELECT title_id, price, advance FROM titles"
Result% = SqlCmd%(SqlConn%, cmd$)

'Send the command to SQL Server and start execution.
Result% = SqlExec%(SqlConn%)
Result% = SqlResults%(SqlConn%)
```

```

'Get the column name, the datatype used by the server, and the maximum
'possible length of the data in the column and print the results.
IF Result% = SUCCEED THEN
    DO UNTIL Result% = NOMOREROWS
        Result% = SqlNextRow%(SqlColumn%)
        IF Result% = NOMOREROWS THEN Exit
    DO
        ELSE
            GetColumnInfo% = SqlGetColumnInfo%(SqlConn%,
            Dim ColDetail as ColumnData, Column%)
            IF GetColumnInfo% = SUCCEED THEN
                ColumnName$ = ColDetail.ColName$
                ColDatatype$ = ColDetail.ColSqlType$
                ColLength& = ColDetail.Collen&
                PRINT "Column name is " + ColumnName$
                PRINT "Datatype of column is " + ColDatatype$
                PRINT "Maximum possible length of data = " + ColLength&
                PRINT
            END IF
        END IF
    LOOP

```

#### **See Also**

[SqlColLen%](#), [SqlColType%](#), [SqlColName\\$](#), [SqlGetAltColInfo%](#)

# SqlOpenConnection%

Opens a login record, establishes a connection to the server, and then deallocates the login record.

## Syntax

**SqlOpenConnection%** ( *server\$, loginid\$, pwd\$, workstation\$, app\$* )

where

*server\$*

Is the name of the SQL Server you want to connect to.

*loginid\$*

Is the login identification number sent to the server. The string can have as many as 30 characters.

*pwd\$*

Is the password sent to the server. The string can have as many as 30 characters.

*workstation\$*

Is the workstation name sent to the server. The string can have as many as 30 characters.

*app\$*

Is the application name sent to the server. The string can have as many as 30 characters.

## Returns

The identifier of a SQL Server connection. If the connection cannot be created, 0 is returned.

## Remarks

**SqlOpenConnection** performs the actions required to open a login record, establish a connection to the server, and then free the memory allocated for the login record. It is the equivalent of combining calls to **SqlLogin%**, **SqlSetLUser%**, **SqlSetLPwd%**, **SqlSetLApp%**, **SqlOpen%**, and **SqlFreeLogin**.

**Important** Do not modify the identifier returned by **SqlOpenConnection** in any way. Modifying the identifier can cause unpredictable results.

## See Also

[SqlFreeLogin](#), [SqlLogin%](#), [SqlOpen%](#), [SqlSetLApp%](#), [SqlSetLPwd%](#), [SqlSetLUser%](#)

# SqlSendCmd%

Sends Transact-SQL text from the command buffer and sets up the statement for processing. This utility combines several functions into a single call.

## Syntax

**SqlSendCmd%** ( *sqlconn%*, *cmd\$* )

where

*sqlconn%*

Is a SQL Server connection. The value of *sqlconn%* is returned by **SqlOpen%**.

*cmd\$*

Is a character string to be copied into the command buffer.

## Returns

SUCCEED (1) or FAIL (0).

## Remarks

**SqlSendCmd%** performs several actions that process the Transact-SQL statements in the command buffer. **SqlSendCmd%** is equivalent to calling **SqlCmd%**, **SqlExec%**, and **SqlResults%**.

**SqlSendCmd%** prepares the command for processing the first set of results but does not retrieve the first data row.

## Example

```
'Put commands into the command buffer.  
cmd$ = "SELECT db_name(dbid), dbid, size FROM sysusages"  
cmd$ = cmd$ + " ORDER BY dbid"
```

```
'Send commands to SQL Server and start execution.  
Result% = SqlSendCmd%(SqlConn%,Cmd$)
```

## See Also

[SqlCmd%](#), [SqlExec%](#), [SqlResults%](#)

# DB-Library for Visual Basic Options

The DB-Library for Visual Basic options described in this appendix give you additional control over SQL Server queries. Design your applications to set and clear these options using **SqlSetOpt%** and **SqlClrOpt%**. Although the Transact-SQL SET statement can set and clear some DB-Library for Visual Basic options, **SqlSetOpt%** and **SqlClrOpt%** provide a uniform interface for all the options. They also mean that your application can use **SqlIsOpt%** to check the status of an option.

Some of the DB-Library for Visual Basic options take parameters, and in these cases the parameters are included in the descriptions below. However, each time you use **SqlSetOpt%**, **SqlClrOpt%**, or **SqlIsOpt%**, you must include both an option and a parameter, whether the option takes a parameter or not.

For example, the SQLROWCOUNT option requires a parameter to set the number of rows to be counted. To set row count to 100, use the following statement:

```
SqlSetOpt%(SqlConn%, SQLROWCOUNT, "100")
```

The SQLARITHABORT option does not require a parameter. To set this option, use an empty string as a parameter, as shown in the following statement:

```
SqlSetOpt%(SqlConn%, SQLARITHABORT, "")
```

For more information on **SqlSetOpt%**, **SqlClrOpt%**, and **SqlIsOpt%**, see [Stored Procedure Functions](#).

## SQLARITHABORT

Terminates a query when an overflow or divide-by-zero error occurs during query execution. If SQLARITHABORT is not set, SQL Server substitutes null values and returns a warning message after the query has been executed. The default setting is off.

## SQLARITHIGNORE

Substitutes null values when an overflow or divide-by-zero error occurs during a query. No warning message is returned. If SQLARITHIGNORE is not set, SQL Server substitutes null values and returns a warning message after the query has been executed. The default setting is off.

## SQLBUFFER

Buffers result rows. SQLBUFFER is required when you use **SqlGetRow%**. You can only set this option using DB-Library for Visual Basic; it cannot be set with the Transact-SQL SET statement. When you set SQLBUFFER, supply a parameter for the number of rows you want buffered. The default setting is 0 (no row buffering).

Parameter	Description
Less than 0	Buffer set to 100 rows.
0	No result rows buffered.
1	Not allowed.
2 - 32, 767	The number of rows to buffer.

For more information about row buffering, see **SqlNextRow%**, **SqlGetRow%**, and **SqlClrBuf** in [Stored Procedure Functions](#).

## SQLCLIENTCURSORS

Forces the use of client cursors. When this option is set, every cursor opened with **SqlCursorOpen%** will be a client cursor. Even when connected to SQL Server 6.0, server cursors will not be used.

This option can be enabled for a client running Windows by placing the line **UseClientCursors=ON** in the **[SQLSERVER]** section of WIN.INI.

## SQLNOAUTOFREE

Causes the command buffer to clear only with a call to **SqlFreeBuf**. When SQLNOAUTOFREE is not set, the first call to **SqlCmd%** after a call to **SqlExec%** or **SqlSend%**, automatically clears the command buffer before new text is entered. The default setting is off.

#### SQLNOCOUNT

Stops returning information about the number of rows affected by each Transact-SQL statement. The default setting is off.

#### SQLNOEXEC

Processes a query through the compile step but does not execute it. You can use this option with SQLSHOWPLAN. Once SQLNOEXEC is set, no subsequent statements are executed until SQLNOEXEC is turned off. The default setting is off.

#### SQLOFFSET

Indicates where SQL Server should return offsets to certain constructs in the query. This option takes a parameter that specifies the particular construct. Valid values of this parameter include:

<i>select</i>	<i>order</i>	<i>procedure</i>
<i>from</i>	<i>compute</i>	<i>execute</i>
<i>table</i>	<i>statement</i>	<i>param</i>

**Note** The value *param* refers to parameters of stored procedures.

For the internal types that correspond to these offset values, see the **SqlGetOff%** function description. Offsets are returned only if the batch contains no syntax errors.

#### SQLPARSEONLY

Checks the syntax of a query and returns error messages to the workstation. The default setting is off.

#### SQLROWCOUNT

Specifies a maximum number of regular rows to be returned for SELECT statements.

SQLROWCOUNT does not limit the number of compute rows returned.

When you set SQLROWCOUNT, supply a parameter for the number of rows you want returned. The default setting is 0, which returns all rows determined by SELECT statements.

Parameter	Description
0	Returns all rows generated by a SELECT statement.
1 - 2, 147, 483, 647	Defines the maximum number of regular rows to be returned for SELECT statements.

This option is different from the other options in that you can turn it off by using **SqlClrOpt%** or by using **SqlSetOpt%** to set it to 0.

#### SQLSHOWPLAN

Generates a description of the processing plan after compilation and continues executing the query. The default setting is off.

#### SQLSTAT

Returns performance statistics (CPU time, elapsed time, I/O) to the workstation after each query. DB-Library for Visual Basic receives these statistics in the form of informational messages, and applications can access them through a user-defined message handler.

When you set SQLSTAT, supply a parameter for the type of performance statistics you want. The default setting is off.

Parameter	Description
IO	Returns statistics about SQL Server's internal I/O: the number of scans, the number of logical reads (pages accessed), and the number of physical reads (disk

accesses) for each table referenced in the query. Also displays the number of pages written for each statement.

**TIME** Returns information about SQL Server parsing, compilation, and execution times. Times are given in milliseconds.

#### SQLSTORPROCID

Sends the stored procedure ID to the workstation before sending rows generated by the stored procedure.

#### SQLTEXTLIMIT

Causes DB-Library for Visual Basic to limit the size of returned text or image values. When setting this option, supply a parameter that is the length, in bytes, of the longest text or image value your application can handle. DB-Library for Visual Basic reads but ignores any part of a text or image value that goes over this limit. In the case of very large text values, it may take some time for the entire text value to be returned over the network. To keep SQL Server from sending this extra text in the first place, use the SQLTEXTSIZE option. The default setting is 4096.

<b>Parameter</b>	<b>Description</b>
0 - 32, 768	Size, in bytes, of the longest text or image value that your application can handle.

#### SQLTEXTSIZE

Limits the size of text or image values SQL Server returns. When setting this option, supply a parameter that is the length, in bytes, of the longest text or image value that SQL Server returns. In applications that allow users to make ad hoc queries, the user can override this option with the Transact-SQL SET TEXTSIZE statement. To set a text limit that the user cannot override, use the SQLTEXTLIMIT option instead. The default setting is 4096.

<b>Parameter</b>	<b>Description</b>
0 - 32, 768	Size, in bytes, of the longest text or image value that SQL Server returns.

# Error Messages

This appendix describes all errors of DB-Library for Visual Basic and their levels of severity. These values are passed to a user-defined error handler. For information on creating an error handler for your application, see [Programming with DB-Library for Visual Basic](#).

The text of the DB-Library for Visual Basic error messages comes from the underlying DB-Library, which is a library for developing client applications in C. As a result, some of the messages are specific to the DB-Library. In these cases, a version of the message that more accurately describes the error for DB-Library for Visual Basic is included in the entry.

The following table lists the DB-Library for Visual Basic errors, providing the error number, the error constant, and the error message for each error. The constants are defined in the VBSQL.BAS file.

Number	Constant	Error message
10000	SQLEMEM	Unable to allocate sufficient memory.
10001	SQLENULL	NULL DBPROCESS pointer encountered. For Visual Basic: A <i>sqlconn</i> identifier with a value of 0 was encountered.
10002	SQLENLOG	NULL LOGINREC pointer encountered. For Visual Basic: A <i>loginrec%</i> with a value of 0 was encountered.
10003	SQLEPWD	Login incorrect.
10004	SQLECONN	Unable to connect: SQL Server is unavailable or does not exist.
10005	SQLEDDNE	DBPROCESS is inactive or not enabled. For Visual Basic: SQL Server connection is dead.
10006	SQLNULLO	Attempt to log in with null LOGINREC. For Visual Basic: Attempt to log in with a <i>loginrec</i> value of 0.
10007	SQLESMMSG	General SQL Server error: Check messages from the SQL Server.
10008	SQLEBTOK	Bad token from SQL Server: datastream processing out of sync.
10009	SQLENSPE	Non-specific general error.
10010	SQLERead	Read from SQL Server failed.
10011	SQLECNOR	Column number out of range.
10012	SQLETSIT	Attempt to call <b>dbtsput</b> with an invalid timestamp. For Visual Basic: Attempt to call <b>SqITsPut%</b> with an invalid timestamp.
10013	SQLEPARM	Invalid parameter in DB-Library function reference.
10014	SQLEAUTN	Attempt to update the timestamp of a table that has no timestamp column.
10015	SQLECOFL	Data-conversion resulted in overflow.
10016	SQLERDCN	Requested data conversion does not exist.
10017	SQLEICN	Invalid <i>computeid</i> or compute column number.
10018	SQLECLOSE	Error in closing network connection.
10019	SQLENTXT	Attempt to get text point/timestamp from a



		non-text column. For Visual Basic: Attempt to get a text identifier or text timestamp from a non-text column.
10020	SQLEDNTI	Attempt to use <b>dbtxtsput</b> to put a new text timestamp into a column whose datatype in neither SQLTEXT nor SQLIMAGE. For Visual Basic: Attempt to use <b>SqlTxTsPut%</b> to put a new text timestamp into a column whose datatype is neither <i>text</i> nor <i>image</i> .
10021	SQLEMTD	Attempt to send too much TEXT data via the <b>dbmoretext</b> call. For Visual Basic: Attempt to send too much text data using <b>SqlMoreText%</b> .
10022	SQLEASEC	Attempt to send an empty command buffer to SQL Server.
10023	SQLENTLL	Name too long for LOGINREC field.
10024	SQLETIME	SQL Server connection timed out.
10025	SQLEWRIT	Write to SQL Server failed.
10026	SQLEMODE	Network connection not in correct mode; invalid SQL Server connection.
10027	SQLEOOB	Error in sending out-of-band data to SQL Server.
10028	SQLEITIM	Illegal timeout value specified.
10029	SQLEDBPS	Maximum number of DBPROCESSes already allocated. For Visual Basic: Invalid or out-of-range parameter to a Visual Basic option.
10030	SQLEIOPT	Attempt to use invalid DBOPTION. For Visual Basic: Attempt to use invalid Visual Basic option.
10031	SQLEASNL	Attempt to set fields in a null LOGINREC. For Visual Basic: Attempt to set fields in a login record with <i>loginrec</i> value of 0.
10032	SQLEASUL	Attempt to set unknown LOGINREC field.
10033	SQLENPRM	NULL parameter not allowed for this DBOPTION. For Visual Basic: This Visual Basic option cannot have a parameter that is an empty string.
10034	SQLEDBOP	Invalid or out of range <b>dbn</b> option parameter. For Visual Basic: Invalid or out-of-range parameter to a Visual Basic option.
10035	SQLENSIP	Negative starting index passed to <b>dbstrcpy</b> . For Visual Basic: Negative starting index passed to <b>SqlStrCpy%</b> .
10036	SQLECNUL	NULL destination variable not allowed. For Visual Basic: You have used 0 as an identifier for a text timestamp.
10037	SQLESEOF	Unexpected EOF from SQL Server.
10038	SQLERPND	Attempt to initiate a new SQL Server operation with results pending.
10039	SQLECSYN	Attempt to convert data stopped by syntax

		error in source field.
10040	SQLENONET	DB-Library network communications layer not loaded.
10041	sqlebytp	Unknown bind type passed to DB-Library function.
10042	sqleabnc	Attempt to bind to a non-existent column.
10043	sqleabmt	User attempted a <b>dbbind</b> with mismatched column and variable types.
10044	sqleabnp	Attempt to bind using NULL pointers.
10045	sqleabncr	Attempt to bind user variable to a non-existent compute row.
10046	slqeaamt	User attempted a <b>dbaltbind</b> with mismatched columns and variable datatypes.
10047	SQLENXID	The server did not grant us a distributed-transaction ID.
10048	sqleifnb	Illegal field number passed to <b>bcp_control</b> .
10049	SQLEKBCO	1000 rows successfully copied to host file.
10050	SQLEBBCI	Batch successfully copied to SQL Server.
10051	SQLEKBCI	1000 rows sent to SQL Server.
10052	SQLEBCWE	I/O error while writing a <b>bcp</b> datafile.
10053	SQLEBCNN	Attempt to copy a null value into a server column that does not accept null values.
10054	SQLEBCOR	Attempt to bulk copy an oversized row to SQL Server.
10055	SQLEBCPI	Call <b>BCPinit</b> before any other <b>BCP</b> routines. For Visual Basic: Call <b>SqlInit\$</b> before any other <b>BCP</b> routines.
10056	sqlebCPn	Use <b>bcp_bind</b> , <b>bcp_colln</b> , and <b>bcp_colptr</b> only after calling <b>bcp_init</b> with the copy direction set to DB_IN.
10057	sqlebCpb	Do not use <b>bcp_bind</b> after <b>bcp_init</b> has been passed a non-null input filename.
10058	SQLEVDPT	For bulk copy, all variable-length data must have either a length-prefix or a terminator specified.
10059	SQLEBIVI	Use <b>bcp_columns</b> and <b>bcp_colfmt</b> only after <b>bcp_init</b> has been passed a valid input file. For Visual Basic: Use <b>SqlBCPColumns%</b> and <b>SqlBCPColfmt%</b> only after <b>SqlInit\$</b> has been passed a valid input file.
10060	SQLEBCBC	Call <b>bcp_columns</b> before <b>bcp_colfmt</b> . For Visual Basic: Call <b>SqlBCPColumns%</b> before <b>SqlBCPColfmt%</b> .
10061	SQLEBCFO	Host files must contain at least one column: <b>bcp</b> .
10062	SQLEBCVH	Call <b>bcp_exec</b> only after <b>bcp_init</b> has been passed a valid host file. For Visual Basic: Call <b>SqlBCPExec%</b> only after <b>SqlBCPInit%</b> has been passed a valid host file.

10063	SQLEBCUO	Unable to open host datafile: <b>bcp</b> .
10064	SQLEBUOE	Unable to open error file: <b>bcp</b> .
10065	SQLEBWEF	I/O error while writing <b>bcp</b> error file.
10066	SQLEBTMT	Attempt to send too much text data with <b>bcp_moretext</b> .
10067	SQLEBEOF	Unexpected EOF encountered in <b>bcp</b> datafile.
10068	SQLEBCSI	Host-file columns can be skipped only when copying into the server.
10069	sqlepnul	NULL program pointer encountered.
10070	SQLEBSKERR	Cannot seek in data file.
10071	SQLEBDIO	Bad bulk-copy direction.
10072	SQLEBCNT	Attempt to use bulk copy with a nonexistent server table.
10073	sqlcmdbp	Attempt to set maximum number of DBPROCESSes lower than 1.
10075	SQLCRSINV	Invalid cursor statement.
10076	SQLCRSCMD	Attempt to call cursor functions when there are commands waiting to be executed.
10077	SQLCRSNOIND	One of the tables involved in the cursor statement does not have a unique index.
10078	SQLCRSDIS	Cursor statement contains one of the disallowed phrases COMPUTE, UNION, FOR BROWSE, or SELECT INTO.
10079	SQLCRSAGR	Aggregate functions are not allowed in a cursor statement.
10080	SQLCRSORD	Only fully keyset driven cursors can have ORDER BY, GROUP BY, or HAVING PHRASES.
10081	SQLCRSMEM	Keyset or window scroll size exceeds the memory limitations of this machine.
10082	SQLCRSBSKEY	Keyset cannot be scrolled backward in mixed cursors with a previous fetch type.
10083	SQLCRSNORE S	Cursor statement generated no results.
10084	SQLCRSVIEW	A view cannot be joined with another table or a view in a cursor statement.
10085	SQLCRSBUFR	Row buffering should not be turned on when using cursor functions.
10086	SQLCRSFROW N	Row number to be fetched is outside valid range.
10087	SQLCRSBROL	Backward scrolling cannot be used in a forward scrolling cursor.
10088	SQLCRSFRAND	Fetch types RANDOM and RELATIVE can only be used within the keyset of keyset driven cursors.
10089	SQLCRSFLAST	Fetch type LAST requires fully keyset driven cursors.
10090	SQLCRSRO	Data locking or modifications cannot be made

		in a READONLY cursor.
10091	SQLCRSTAB	Table name must be determined in operations involving data locking or modifications.
10092	SQLCRSUPDTA B	Update or insert operations using bind variables require single table cursors.
10093	SQLCRSUPDNB	Update or insert operations cannot use bind variables when binding type is NOBIND.
10094	SQLCRSVIIND	The view used in the cursor statement does not include all the unique index columns of the underlying tables.
10095	SQLCRSNOUP D	Update or delete operation did not affect any rows.
10096	SQLCRSOS2	Cursors are not supported for this server.
10097	SQLEBCSA	The BCP hostfile %s contains only %ld rows. Skipping all of these rows is not allowed.
10098	SQLEBCRO	Data locking or modifications cannot be made in a READONLY cursor.
10099	SQLEBCNE	The table %s contains only %ld rows. Copying up to row %ld is not possible.
10100	SQLEBCSK	The table %s contains only %ld rows. Skipping all of these rows is not allowed.
10101	SQLEUVBF	Attempt to read unknown version of <b>bcp</b> format file.
10102	SQLEBIHC	Incorrect host-column number found in <b>bcp</b> format file.
10103	SQLEBWFF	I/O error while reading <b>bcp</b> format file.
10104	SQLNUMVAL	The data stored in the DBNUMERIC/DBDECIMAL structure is invalid.
10105	SQLEOLDVR	The SQL Server's TDS is obsolete with this version of DB-Library.

# Error-Severity Levels

The following table lists the level of severity of Visual Basic errors and the corresponding constants. The constants are defined in the VBSQL.BAS file.

Severity level	Constant	Description
1	EXINFO	Informational non-error.
2	EXUSER	User error.
3	EXNONFATAL	Nonfatal error.
4	EXCONVERSION	Error in Visual Basic data conversion.
5	EXSERVER	Server returned an error flag.
6	EXTIME	Exceeded timeout period while waiting for a response from SQL Server. The SQL Server connection is still alive.
7	EXPROGRAM	Coding error in user program.
8	EXRESOURCE	Running out of resources. The SQL Server connection may be inactive.
9	EXCOMM	Failure in communication with SQL Server. The SQL Server connection is inactive.
10	EXFATAL	Fatal error. The SQL Server connection is inactive.
11	EXCONSISTENCY	Internal software error. Notify your primary support provider.

