Programming DB-Library for C

Microsoft® SQL Server[™] is a powerful structured query language (SQL) database server. With local area network (LAN) software, SQL Server allows clients running the Microsoft Windows NT® 4.0, Microsoft Windows®, or Microsoft MS-DOS® operating systems to access its services. Developers can write applications for SQL Server by programming with DB-Library for C, the C-language version of the communication library for SQL Server.

DB-Library for C is an application programming interface (API) consisting of C functions and macros that allow an application to interact with SQL Server. Included are functions that send Transact-SQL statements to SQL Server and functions that process the results of those statements. Other functions handle errors and convert data.

DB-Library for C offers a rich set of functions for:

- Opening connections.
- Formatting queries.
- Sending query batches to the server and retrieving the resulting data.
- Using scrollable cursors.
- Bulk-copying data from files or program variables to and from the server.
- Controlling two-phase commit operations between several participating SQL Servers.
- Executing stored procedures or remote stored procedures.

These functions allow the application developer extremely fine-grained control

of data flow back and forth between the client and the server. DB-Library supports multiple environments. C programmers can choose from among:

- A multithread dynamic-link library (DLL) for the Windows NT 4.0 operating system.
- Medium and large model static-link libraries for MS-DOS for both Microsoft and Borland compilers.
- A DLL for the Windows operating system.

Getting Started with DB-Library for C

The Microsoft DB-Library for C application programming interface (API) is a set of C functions used to create client applications that interact with Microsoft® SQL ServerTM.

You can use DB-Library for C to build applications for the Microsoft Windows NT® 4.0 operating system and to port Microsoft Windows®-based applications to a Windows NT 4.0 system.

DB-Library for C Syntax Conventions

The DB-Library for C syntax diagrams use these conventions.

Convention	Used for
UPPERCASE	Transact-SQL keywords, DB-Library data types, data
	structures, and parameter settings.
Italic	User-supplied parameters.
Monospace	Program code.
Bold	Functions, C data types, C statements, database names, table names, column names, index names, stored procedures, utilities, and text that must be typed exactly as shown.

System Requirements for DB-Library for C

To develop applications using DB-Library for C, you must meet the Microsoft® SQL ServerTM client system requirements for the operating system you are using. For information about the system requirements for SQL Server, see *Installing SQL Server*.

Windows NT

To create Microsoft® Win32® DB-Library applications for Microsoft Windows NT® 4.0, you need the files provided with this release, as well as the following software:

• Microsoft Windows NT Workstation version 3.5 or later (version 3.51 or later is recommended)

Or

Microsoft Windows NT Server version 3.5 or later (version 3.51 or later is recommended)

• Microsoft Visual C++® version 2.0 or later, or a 100-percent compatible compiler and linker

DB-Library for Windows NT 4.0 is available for computers with the Intel® 32bit x86-based microprocessor architectures:

Windows 95 and Windows 98

To create Microsoft Win32 DB-Library applications for Microsoft Windows® 95 and Windows 98, you need the same DB-Library files you use for Microsoft Windows NT 4.0, as well as the following software:

- Microsoft Windows 95 and Windows 98
- Microsoft Visual C++® version 2.0 or later, or a 100-percent compatible compiler and linker

Windows

To create Win16 DB-Library applications for Microsoft Windows, you need the files provided with this release, as well as the following software:

• Microsoft Windows version 3.11 or later

Or

Microsoft Windows for Workgroups version 3.11 or later

• Microsoft Visual C++ for Windows version 1.0 or later

Or

Borland C++ version 3.1 or later

MS-DOS

To create 16-bit DB-Library applications for Microsoft MS-DOS®, you need the files provided with this release, as well as the following software:

- MS-DOS version 5.0 or later
- Microsoft Visual C++ for Windows version 1.0 or later

Or

Borland C++ version 3.1 or later

Installing DB-Library for C

SQL Server Setup installs the following groups of files.

 $In \Mssql7\DevTools\Include$

File	Description
Sqldb.h	DB-Library function prototypes.
Sqlfront.h	DB-Library type and macro definitions.

In Mssql7DevToolsLib

File	Description
Bldblib.lib	Borland large-model DB-Library static library for
	Microsoft® MS-DOS®.
Bmdblib.lib	Borland medium-model DB-Library static library for
	MS-DOS.
Msdblib3.lib	DB-Library import library for Microsoft Windows®.
Ntwdblib.lib	DB-Library import library for Microsoft Win32 [®] .
Rldblib.lib	Large-model DB-Library static library for MS-DOS.
Rmdblib.lib	Medium-model DB-Library static library for MS-DOS.
W3dblib.lib	Old DB-Library import library for Windows. (Use
	Msdblib3.lib, not this file.)

 $In \Mssql7\DevTools\Samples\Dblib\C$

File	Description
Readme.txt	Summary of sample programs available in each
	directory.

Determining the Version of DB-Library for C

For Microsoft® Windows® and Microsoft Windows NT® 4.0, run the SQL Server Client Network Utility to determine which version of DB-Library is installed on your computer. This utility searches the path for the appropriate DLL to load (the path begins with the current directory). Run the utility from the same working directory as your application.

For Microsoft MS-DOS[®], include a call to **dbinit** in your DB-Library for C application to determine which version of DB-Library for C is installed. This function returns the version number of DB-Library for C. It is a good idea to display this information in all your DB-Library applications.

Finding Further Information

Information about DB-Library for C and Microsoft® SQL Server[™] is available from the following sources, as well as from the Microsoft Development Library and Microsoft TechNet.

The Microsoft Developer Network provides in-depth information for developers. With Level 1 of the Developer Network, you get four quarterly releases of the Development Library, packed with inside information about programming for Microsoft® Windows® and Microsoft Windows NT® 4.0, plus other program benefits. With Level 2 of the Developer Network, you'll receive all the Level 1 benefits plus quarterly releases of the Development Platform, containing the latest Windows and Windows NT 4.0 Workstation operating systems and APIlevel SDKs and DDKs from Microsoft. For ordering information, call (800) 759-5474.

Microsoft TechNet is the front-line resource for fast, complete answers to technical questions on Microsoft desktop and system products. Through two monthly compact discs, you get everything from crucial data on client/server and workgroup computing, systems platforms, and database products to the latest applications support for Microsoft Windows and the Apple Macintosh®. For ordering information, call your reseller or (800) 344-2121.

Programming for SQL Server

There are several methods of programming Microsoft[®] SQL Server[™] applications.

Transact-SQL

Transact-SQL is the SQL Server-enhanced version of the SQL database language. Client applications use Transact-SQL to communicate with SQL Server. Transact-SQL provides statements for creating and manipulating database objects, and for inserting, updating, and selecting data. The Transact-SQL enhancements include data integrity features and stored procedures (compiled static SQL statements). Stored procedures allow much of an application's processing logic to be shifted from the client application to SQL Server. Stored procedures can contain most Transact-SQL statements, including Transact-SQL control-of-flow statements. Furthermore, stored procedures are precompiled, so the statements don't have to be parsed each time the procedure is executed.

DB-Library for C Applications

DB-Library for C enables the database to become an integral part of an application. Transact-SQL statements can be incorporated into the application, allowing the application to retrieve and update values from a database. Through DB-Library for C, values from the database can be placed in program variables for manipulation by the application. Conversely, values in program variables can be inserted into the database.

Although DB-Library for C contains many functions, giving the application much control over its interaction with SQL Server, most applications require only a few functions. The actual process of connecting with SQL Server, sending Transact-SQL statements to SQL Server, and manipulating the resulting data is straightforward.

DB-Library Compared to Embedded SQL

DB-Library differs distinctly from the Embedded SQL for C type of language

interface:

- The DB-Library interface does not require a language precompiler.
- DB-Library consists of C functions that are not preprocessed into an intermediate form.

Avoiding preprocessing can make database applications more straightforward to write and to debug. However, for programmers new to developing SQL Server applications or who are familiar with other database platforms, Embedded SQL offers a simpler programming interface. Tools for building Embedded SQL applications are available separately from Microsoft.

DB-Library for C Compared to Microsoft ODBC

In addition to DB-Library, SQL Server supports the Microsoft Open Database Connectivity (ODBC) call-level API. ODBC enables applications for Microsoft Windows NT® 4.0, Microsoft Windows®, Windows 95, and Windows 98 to access multiple data sources, including a wide range of relational databases and local indexed sequential access method (ISAM) data. In most cases, DB-Library and ODBC are equally capable of creating SQL Server client applications. ODBC enables applications to access more data sources. DB-Library provides equivalent functionality in multiple client environments, including Microsoft MS-DOS®. In addition, DB-Library provides specialized support for SQL Server, such as bulk copy, two-phase commit, and text operations.

Tools for developing ODBC applications are available in the Microsoft ODBC Software Development Kit, which is part of the Microsoft Developer Network Level II. The ODBC driver for SQL Server is included with SQL Server.

Communicating with SQL Server

DB-Library functions communicate with Microsoft® SQL Server[™] 2000 through the DBPROCESS structure. The **dbopen** function allocates and initializes a DBPROCESS when it logs on to SQL Server. DBPROCESS serves as the connection between the application and SQL Server. Most DB-Library functions require DBPROCESS as the first parameter. An application can have more than one DBPROCESS if, for instance, it needs to update a database while still processing the results of an earlier query. Each DBPROCESS is completely independent of any other.

Another structure, LOGINREC, allocated by **dblogin**, is the login record that contains information the **dbopen** function uses to log on to SQL Server. It contains typical login information, such as the username and password. This information is also specified through DB-Library for C functions.

Note With the Microsoft Windows NT® 4.0, Microsoft Windows® 95 and Windows 98 operating systems, each connection is a separate execution thread that is spawned when the connection to SQL Server is established with **dbopen** and terminated when the connection is closed using **dbclose**.

DB-Library for C and Net-Library Interaction

When a call is made to open a connection to Microsoft® SQL ServerTM 2000 (**dbopen**), DB-Library determines which client Net-Library should be loaded to communicate with SQL Server.

Net-Libraries are linked dynamically at run time. With the Microsoft Windows NT® 4.0, Microsoft Windows® 2000, Microsoft Windows 95, and Microsoft Windows 98 operating systems, Net-Libraries are implemented as dynamic-link libraries (DLLs), and multiple Net-Libraries can be loaded simultaneously. With the Microsoft MS-DOS® operating system, Net-Libraries are implemented as terminate-and-stay-resident (TSR), and only one can be loaded at any given time.

Net-Library Architecture

Microsoft® SQL Server[™] 2000 Net-Library architecture for client/server applications is based on the Net-Library concept that abstracts the client and server applications from the underlying network protocols being used.

Tabular Data Stream (TDS) is the data stream protocol used by SQL Server to transfer requests and responses between the client and the server. TDS is a logical data stream protocol and must be supported by a physical network interprocess communication mechanism (IPC). The Net-Library architecture provides a method of sending TDS across a physical network connection, and it provides a transparent interface to DB-Library for C.

Using Examples with pubs

Throughout this documentation, examples that use the **pubs** database are based on the U.S. English version of **pubs**. Your system must have access to the **pubs** database to run the sample programs. If you have a localized version of Microsoft® SQL Server[™] 2000 and want to try the examples, drop the localized version of **pubs** and install the U.S. English version by running Instpubs.sql. The syntax is as follows:

isql /Usa /Ppassword /Sserver -i\sql\install\instpubs.sql

The **pubs** database is installed in the \Mssql7\Data directory.

Programming with DB-Library for C

Programming with DB-Library for C typically involves the following steps:

- 1. Connect to Microsoft[®] SQL Server[™] 2000.
- 2. Put Transact-SQL statements into a buffer and send them to SQL Server.
- 3. Process the results, if any, returned from SQL Server, one statement at a time and one row at a time. You can put the results into program variables, where the application can manipulate them.
- 4. Handle DB-Library errors and SQL Server messages.
- 5. Disconnect from SQL Server.

The following example shows the basic framework of many DB-Library for C applications. The application connects to SQL Server, sends a Transact-SQL SELECT statement to SQL Server, and processes the set of rows resulting from the SELECT statement.

For more information about defining the target operating system prior to compiling your application, see <u>Building Applications</u>.

#define DBNTWIN32
#include <stdio.h>
#include <windows.h>
#include <sqlfront.h>
#include <sqldb.h>

// Forward declarations of the error handler and message handler. int err_handler(PDBPROCESS, INT, INT, INT, LPCSTR, LPCSTR); int msg_handler(PDBPROCESS, DBINT, INT, INT, LPCSTR, LPCST

LPCSTR, DBUSMALLINT);

```
main()
{
    PDBPROCESS dbproc; // The connection with SQL Server.
    PLOGINREC login; // The login information.
    DBCHAR name[100];
    DBCHAR city[100];
```

// Install user-supplied error- and message-handling functions.
dberrhandle (err_handler);
dbmsghandle (msg_handler);

```
// Initialize DB-Library.
dbinit ();
```

```
// Get a LOGINREC.
login = dblogin ();
DBSETLUSER (login, "my_login");
DBSETLPWD (login, "my_password");
DBSETLAPP (login, "example");
```

// Get a DBPROCESS structure for communication with SQL Serve: dbproc = dbopen (login, "my_server");

// Retrieve some columns from the authors table in the
// pubs database.

// First, put the command into the command buffer.
dbcmd (dbproc, "SELECT au_lname, city FROM pubs..authors");
dbcmd (dbproc, "WHERE state = 'CA' ");

// Send the command to SQL Server and start execution.

```
dbsqlexec (dbproc);
```

```
// Process the results.
  if (dbresults (dbproc) == SUCCEED)
  {
    // Bind column to program variables.
    dbbind (dbproc, 1, NTBSTRINGBIND, 0, name);
    dbbind (dbproc, 2, NTBSTRINGBIND, 0, city);
    // Retrieve and print the result rows.
    while (dbnextrow (dbproc) != NO_MORE_ROWS)
     {
       printf ("%s from %s\n", name, city);
     }
  }
  // Close the connection to SQL Server.
  dbexit ();
  return (0);
}
int err_handler (PDBPROCESS dbproc, INT severity,
  INT dberr, INT oserr, LPCSTR dberrstr, LPCSTR oserrstr)
{
  printf ("DB-Library Error %i: %s\n", dberr, dberrstr);
  if (oserr != DBNOERR)
  {
    printf ("Operating System Error %i: %s\n", oserr, oserrstr);
  }
  return (INT CANCEL);
}
```

```
int msg_handler (PDBPROCESS dbproc, DBINT msgno, INT msgstat
INT severity, LPCSTR msgtext, LPCSTR server,
LPCSTR procedure, DBUSMALLINT line)
{
    printf ("SQL Server Message %ld: %s\n", msgno, msgtext);
    return (0);
}
```

This example illustrates features common to most DB-Library for C applications, including:

header files

All source files that contain calls to DB-Library functions require two header files, Sqlfront.h and Sqldb.h.

Before including the Sqlfront.h and Sqldb.h files, define the target operating system with **#define**:

- DBMSDOS (for Microsoft MS-DOS®)
- DBMSWIN (for 16-bit Microsoft Windows®)
- DBNTWIN32 (for 32-bit Windows 95 and Microsoft Windows NT® 4.0)

An alternative is to put DBMSDOS, DBMSWIN, or DBNTWIN32 on the compilation command lines. For more information, see the examples in "Include Files", in <u>Building Applications</u>.

For Windows, Windows 95, and Windows NT 4.0, you must include Windows.h before including the Sqlfront.h and Sqldb.h files.

Include Sqlfront.h before Sqldb.h. Sqlfront.h defines symbolic constants, such as function return values and the exit values STDEXIT and ERREXIT. These exit values can be used as the parameter for the C standard library function **exit**. The exit values are defined appropriately for the operating system running the

application. The Sqlfront.h file also includes type definitions for data types that can be used in program variable declarations. These data types are described in <u>DB-Library for C Data types</u>.

The Sqldb.h file contains additional type definitions and DB-Library function prototypes, most of which are meant to be used only by the DB-Library functions. They should not be accessed directly by the program. To ensure compatibility with future releases of DB-Library, use the contents of Sqldb.h only as documented here.

dberrhandle and dbmsghandle

The first of these DB-Library functions, **dberrhandle**, installs a usersupplied error-handling function, which is called automatically whenever the application encounters a DB-Library error. Similarly, **dbmsghandle** installs a message-handling function, which is called in response to informational or error messages returned from SQL Server. The error- and message-handling functions are user-supplied. It is strongly recommended that users supply error-processing functions.

dblogin

Supplies a LOGINREC structure, which DB-Library uses to log on to SQL Server. Two functions set entries in the LOGINREC. **DBSETLPWD** sets the password that DB-Library uses when logging in. **DBSETLAPP** sets the name of the application, which appears in the SQL Server **sysprocess** table. Certain functions set other aspects of the LOGINREC, which contains defaults for each value they set.

dbopen

Opens a connection between the application and SQL Server. It uses the LOGINREC supplied by **dblogin** to log on to the server. It returns a DBPROCESS structure, which serves as the conduit for information between the application and the server. After this function has been called, the application is connected with SQL Server and can send Transact-SQL statements to SQL Server and process the results. Simultaneous transactions must each have a distinct DBPROCESS. Serial transactions can use the same DBPROCESS.

dbcmd

Fills the command buffer with Transact-SQL statements, which can then be sent to SQL Server. Each call to **dbcmd**, after the first, adds the supplied text to the end of any text already in the buffer. The programmer must supply necessary blanks between words, such as the space between the quotation mark and the word WHERE in the second **dbcmd** call in the example:

dbcmd(dbproc, " WHERE state = 'CA' ");

Although multiple statements can be included in the buffer, this example only shows how to send and process a single statement. DB-Library allows an application to send multiple statements (called a *command batch*) to SQL Server and process each statement's set of results separately.

dbsqlexec

Executes the command buffer; that is, it sends the contents of the buffer to SQL Server, which parses and executes the commands. This function causes DB-Library to wait until SQL Server has completed execution of the query. To avoid this delay, you can call **dbsettime** to set the DB-Library time-out, or you can use **dbsqlsend**, **dbdataready**, and **dbsqlok** (instead of **dbsqlexec**) to retain control while SQL Server is busy.

dbresults

Gets the results of the current Transact-SQL statement ready for processing. After **dbresults** returns SUCCEED, column meta data for the current result set is available. Your application should call **dbresults** until it returns NO_MORE_RESULTS. If your program fails to do this, the DB-Library error message 10038 "Results Pending" occurs the next time that DBPROCESS is used.

dbbind

Binds result columns to program variables. In the example, the first call to **dbbind** binds the first result column to the *name* variable. In other words, when the program reads a result row by calling **dbnextrow**, the contents of the first column in the result row are placed in the *name* variable. The data type of the binding is NTBSTRINGBIND, one of several binding types available for character data. The second call binds the second result column to the *city* variable.

dbnextrow

Reads a row and places the results in the program variables specified by the earlier **dbbind** calls. Each successive call to **dbnextrow** reads another result row until the last row has been read and NO_MORE_ROWS is returned. Processing of the results must take place inside the **dbnextrow** loop. This is because each call to **dbnextrow** overwrites the previous values in the bound program variables.

dbexit

Closes all SQL Server connections and frees all DBPROCESS structures created because of the application. It is usually the last DB-Library function in the program.
DB-Library for C Data Types

DB-Library defines data type tokens for Microsoft® SQL Server[™] data. These data type constants begin with "SQL" (for example, SQLINT4, SQLMONEY, or SQLCHAR). DB-Library also provides type definitions for use in program variable declarations. These type definitions begin with the prefix "DB" (for example, DBINT, DBCHAR, or DBMONEY). By using them, you ensure that your program variables will be compatible with SQL Server data types. For more information about a list of SQL Server data types and the DB-Library program variable types, see Using DB-Library for C Data Types.

The **dbconvert** function provides a way to convert data from one SQL Server data type to another. It supports conversion between most data types. Because SQL Server data types correspond directly to the DB-Library data types, you can use **dbconvert** widely within your application.

The functions that bind SQL Server result columns to program variables **dbbind** and **dbaltbind** also provide type conversion.

When the operators SUM and AVG are applied to any column with a small data type (such as **smalldatetime, real, smallmoney, smallint**, or **tinyint**), the server returns results in the larger data type. For example, the table **sales** has a **smallmoney** column named **price** and the following query is executed on this table:

SELECT price FROM sales

COMPUTE SUM(price), AVG(price), MIN(price), MAX(price)

The returned values *price*, MIN(*price*), and MAX(*price*) are of the data type **smallmoney**; AVG(*price*), and SUM(*price*) are of the data type **money**.

Finding SQL Servers on the Network

You can use the **dbserverenum** function to obtain the names of servers running Microsoft® SQL Server[™] 2000 either locally or over the network. For network searches, the Net-Library DLL used must support the **dbserverenum** function.

Resolving Server Names

DB-Library resolves server names differently depending on the client platform. It is recommended that you use SQL Server Client Network Utility to configure server name and Net-Library connection information.

Windows NT-based Clients

For Microsoft® Windows NT® 4.0- based clients, when **dbopen** is called with the name of a server running Microsoft SQL Server[™] 2000 to connect to, DB-Library uses the configuration information in the following subtree of the Windows NT Registry:

HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ MSSQLServer\ Client\ ConnectTo

The entries of the Registry are values in the following format:

value:REG_SZ:Net-Library_Name[,network_specific_parameters]

Windows-based Clients

For Microsoft® Windows®-based clients, when **dbopen** is called with the name of a server running Microsoft SQL ServerTM 2000 to connect to, DB-Library uses the configuration information in the [**SQLSERVER**] section of the Win.ini file. The entries of the [**SQLSERVER**] section of Win.ini have the following format:

logical_name = Net-Library_Name[,network_specific_ parameters]

MS-DOS-based Clients

With Microsoft® MS-DOS®, only one Net-Library TSR can be loaded, so there is no .ini configuration. Instead, MS-DOS environment variables are used to specify any network-specific connection information. Environment variables have the following format:

logical_name = network_specific_ parameters

Before loading the Net-Library TSR, the environment variables containing connection information must be set. If the *servername* parameter passed from **dbopen** corresponds to a currently set environment variable, the Net-Library uses the information in the environment string to determine server location and network-specific information parameters, if present. If no environment variable matches the *servername* passed from **dbopen**, the Net-Library uses the *servername* parameter passed from **dbopen**.

Results Processing

When **dbsqlexec** or **dbsqlok** returns SUCCEED, it indicates that Microsoft® SQL Server[™] 2000 has successfully executed a command batch. At that point, the application must process any results. The SELECT statement and EXECUTE statements that contain SELECT statements both return result rows. INSERT statements and most other Transact-SQL statements also return data needed by DB-Library.

The following list describes the two types of result rows:

- Regular rows are generated from columns in a SELECT statement's select list.
- Compute rows are generated from columns in a SELECT statement's COMPUTE clause.

Because these two types of rows contain different data, the application must process them separately.

The result for each statement in a batch is returned to the application separately. Within each statement's set of results, the result rows are processed one at a time.

The results for each statement in the batch must be set up separately with **dbresults**.

Getting Result Data

The simplest way to get result data is to bind result columns to program variables with **dbbind** and **dbaltbind**. Then, when a result row is accessed, DB-Library automatically places copies of the columns' data into the program variables to which they are bound.

You also can access a result column's data directly with **dbdata** and **dbadata**, which return pointers to the data. These functions are frequently used with **dbdatlen** and **dbadlen**, which return the length of the data. The **dbdata** and **dbadata** functions have the advantage of providing access to the actual data, not a copy of the data.

Binding of columns to variables must take place after the call to **dbresults** but before the first call to **dbnextrow**. No such preliminary step is needed when results will be directly accessed with **dbdata** or **dbadata**.

Accessing Result Rows

After **dbresults** has returned SUCCEED and any binding of columns to variables has been specified, the application is ready to process the results. The first step is to make the result rows available to the application. The **dbnextrow** function accomplishes this. Each call to **dbnextrow** reads the next row returned from Microsoft® SQL Server[™] 2000. The row is read directly from the network. As an alternative, you can do the same thing through cursors, often more simply and with more power. For details, see <u>Bulk-Copy Functions</u>.

After a row has been read in by **dbnextrow**, the application can perform any processing desired on the data in the row. If the result columns have been bound to program variables, the data in the row will have been automatically copied into the variables. Alternatively, the data is accessible through **dbdata** or **dbadata**.

Rows read in by **dbnextrow** can be automatically saved in a row buffer, if desired. This is accomplished by setting the DBBUFFER option. Without row buffering, each row must be processed as it is read in by **dbnextrow** because the next call to **dbnextrow** will overwrite the previously read row. If row buffering has been turned on, the rows are added to a row buffer as they are read in by **dbnextrow**. Row buffering allows the application to skip around in the buffer and return to previously read rows.

After all result rows have been returned, the final call to **dbnextrow** returns the NO_MORE_ROWS indication. You must continue calling **dbnextrow** until it returns NO_MORE_ROWS, even if you know that the query produces only one result row.

Also available are functions that print result rows in a default format. They are often used for debugging. (These are not available with the Microsoft Windows® operating system.)

DB-Library processes results one statement at a time, although multiple statements can be sent. When all the results for one statement have been read, **dbresults** must be called again to set up the results for the next statement in the command buffer and one final time to return NO_MORE_RESULTS.

Information covering several areas, including regular result columns, compute

result columns, row buffers, and the command state, can be retrieved from the DBPROCESS structure. Regular result columns correspond to columns in the SELECT statement's select list, and compute result columns correspond to columns in the SELECT statement's optional COMPUTE clause.

Regular Result Column Information

These functions can be called after **dbresults** returns SUCCEED.

Function	Description
Dbcollen	Returns the maximum length of a regular column's data.
dbcolname	Returns the name of a regular result column.
Dbcoltype	Returns the Microsoft® SQL Server [™] data type for a regular result column.
dbcolutype	Returns the user-defined data type for a regular result column.
Dbdatlen	Returns the actual length of a regular column's data. This function is often used together with dbdata . The value returned by dbdatlen can differ for each regular row read by dbnextrow .
dbnumcols	Determines the number of columns in the current result set.

Compute Result Column Information

These functions can be called after **dbnextrow** or **dbgetrow** returns a compute ID.

Function	Description
dbadlen	Returns the actual length of a compute column's data. This function is often used together with dbadata . The value returned by dbadlen can differ for each compute row read by dbnextrow .
dbaltcolid	Returns the column ID for a compute column.
dbaltlen	Returns the maximum length for a compute column's data.
dbaltop	Returns the type of aggregate operator for a compute column.
dbalttype	Returns the data type for a compute column.
dbaltutype	Returns the user-defined data type for a compute column.
dbbylist	Returns the bylist for a compute row.
dbnumalts	Returns the number of columns in a compute row.
dbnumcompute	Returns the number of COMPUTE clauses in the current set of results.

When to Process Results

It is critical that all result sets are processed completely in a timely manner. The result sets need to be processed to avoid problems with subsequent SQL Server queries and to avoid concurrency issues with SQL Server resources.

In most cases, the return code from **dbsqlok** or **dbsqlexec** should be ignored. If you send the following batch and the INSERT statement fails due to a duplicate key, a severity 14 error is generated; however, the batch continues.

INSERT INTO tblTest VALUES(1) SELECT @@VERSION

The **dbsqlok** and **dbsqlexec** calls check only the success of the first command. If you do not call **dbresults**, you will not process the SELECT statement results and can get result pending errors.

The following are the most common problems your application may encounter if result sets are not handled immediately and completely:

 If all result sets are not processed completely and you attempt to send another query to SQL Server using the same connection, you will receive DB-Library error 10038: Attempt to initiate a new SQL Server operation with results per

DB-Library prevents you from sending additional queries if there are results from a previous query that need to be handled. For more information, see <u>When and How to Use dbcancel()</u>.

• If a query is issued to SQL Server and the results are not handled immediately, you may be holding locks and reducing concurrency on your server.

For example, you issue a query that requires rows from two pages to populate your result set. SQL Server parses, compiles, and runs the query. This means that shared locks are placed on the two pages that contain the rows needed to satisfy your query. Further, not all rows fit onto one SQL Server Tabular Data Stream (TDS) packet. TDS packets are filled and sent to the client. If all rows from the first page fit on the TDS packet, SQL Server releases the shared lock on that page, but leaves a shared lock on the second page. SQL Server then waits for the client to request more data (this is done using **dbnextrow/dbresults**, **SQLNextRow/SQLResults**, **FetchLast/FetchFirst**, and so on).

This means that the shared lock is held until the client requests the rest of the data. Other processes requesting data from the second page may be blocked. For more information, see <u>Troubleshooting Locking</u>.

Following is an example of how to handle all result sets correctly.

```
BOOL bMoreResults = TRUE;
BOOL bMoreRows = TRUE;
RETCODE dbRC = SUCCEED;
```

```
//
// Send query
```

```
.
.
.
.
//
// Process *all* results
bMoreResults = TRUE
while(bMoreResults)
{
    switch(dbRC = dbresults(pdbproc))
    {
    case SUCCEED:
```

```
bMoreRows = TRUE;
```

```
while(bMoreRows)
 ł
  switch(dbRC = dbnextrow(pdbproc))
  ł
   case REG_ROW:
    // Handle regular row
    break;
   case NO MORE ROWS:
    bMoreRows = FALSE;
    // All rows in this result set handled
    break;
   case BUF_FULL:
    // Handle full buffer when using row buffering
    break;
   case FAIL:
    // Any error processing desired
    bMoreRows = FALSE;
    break;
   default:
    // Handle compute row
    break;
  }
 }
 break;
case NO MORE RESULTS:
```

```
bMoreResults = FALSE; // All result sets handled break;
```

case FAIL:

// Any error processing desired
// The current command has returned an error
// Could be a nonfatal error
bMoreResults = TRUE;
break;

```
case NO_MORE_RPC_RESULTS:
    // Extract stored procedure return information
    break;
```

default: bMoreResults = FALSE; // unknown

break;

}

} // while(bMoreResults && FALSE == DBDEAD(pdbproc)

When and How to Use dbcancel()

The **dbcancel()** function is often used in applications when it should not be. A large percentage of common DB-Library or VBSQL programming problems stem from applications that misuse this API call.

When retrieving results or after sending a Transact-SQL batch to the server, it is considered good DB-Library programming practice to process all results until there are no more results, and to process all rows until there are no more rows. For example, you should call **dbresults()** and **dbnextrow()** like this:

```
while ((result_code = dbresults(dbproc)) != NO_MORE_RESULTS)
{
    while (dbnextrow(dbproc) != NO_MORE_ROWS)
}
```

Not calling **dbresults()** and **dbnextrow()** as above often causes application problems that may not become evident until later in development or testing when corrective action is costly.

If an application is required to access only *x* number of rows from a result set, it is recommended that the SET ROWCOUNT *x* Transact-SQL statement be used instead of calling **dbnextrow** *x* times followed by a call to **dbcancel()**. There are several reasons for not using **dbcancel()** as part of standard results processing routines.

When Not to Use dbcancel()

dbcancel() will not cancel, roll back, or commit a user-defined transaction. All exclusive locks acquired within the user-defined transaction will be retained even after issuing **dbcancel()** because the user-defined transaction is still active. This can cause blocking and other difficult concurrency problems. When issuing **dbcancel()** within a user-defined transaction, the programmer should ensure that the transaction is either explicitly committed or rolled back.

When operating in this context, it is easy to miss that an application is actually in the middle of a user-defined transaction that was never committed or rolled back.

In addition, there are also some variations of how **dbcancel()** or the attention signal that it generates are implemented from platform to platform.

Note Microsoft® SQL Server[™] implementations of **dbcancel()** are constant across all SQL Server platforms. Compatibility issues arise primarily between different Sybase and SQL Server implementations. These differences often stem from transport related issues such as how one vendor's TCP/IP out-of-band data is implemented and how it interacts with another vendor's TCP/IP out-of-band data implementation. The net results of which can cause portability problems in DB-Library applications using **dbcancel()**.

When to Use dbcancel()

dbcancel() should be used in cases where the user needs to regain control of an application. In this environment, a programmer can handle the exception of a user issuing a **dbcancel()** in an environment where **dbcancel()** does not work by responding with a message such as "**dbcancel()** not supported on this platform!". A programmer may also want to use **dbcancel()** in case of a DB-Library command failure as part of a clean up procedure for the affected DB-Lib processes.

See Also

dbcanceldbprocmsghandledbexitdbsetmaxprocsdbinitProgramming DB-Library for Cdbprocerrhandle

Error and Message Handling

DB-Library for C applications deal with errors and messages: Messages generated by DB-Library for C functions are called *errors* and those from Microsoft® SQL Server[™] 2000 are known as *messages*. SQL Server can return both informational and error messages to the application. In addition, DB-Library for C has its own set of possible warnings and errors. A list of SQL Server messages appears in the **sysmessages** table. For more information about a list of DB-Library for C errors, see <u>Error Messages</u>.

DB-Library for C easily centralizes error and message handling. With **dbmsghandle** and **dberrhandle**, you can install your own message- and errorhandling functions. When a message or error occurs, DB-Library for C automatically calls the appropriate user-supplied function, providing information to that function on the nature of the message or error, thus the error- and message-handling logic can be assigned to two functions in your application.

Browse Mode

By creating temporary tables, browse mode lets you scan database rows and update their values one row at a time. This feature uses optimistic concurrency control, which holds no locks while you are accessing data. Browse mode requires several steps because it transfers each row from the database into program variables before browsing and updating it.

Because a row being browsed is just a copy residing in program variables, rather than the actual row in the database, the application must ensure that changes to the variables' values reliably update the original database row. In particular, in multiuser situations, the application must ensure that updates to the database made by one user do not overwrite updates made by another. This becomes an issue whether the application selects one row or several rows at a time from the database. A **timestamp** column in database tables that can be browsed lets you regulate this type of multiuser updating.

Browse-mode functions also allow an application to update ad hoc queries (queries made while an application is running). Several of these functions return information that an application can use to examine the structure of a complicated ad hoc query to update the underlying database tables.

Conceptually, browse mode does the following:

- 1. Issues a SELECT statement.
- 2. Fetches rows.
- 3. If any changes must be made, constructs an UPDATE statement and issues it.

In an application, implement these steps as follows:

1. Execute a SELECT statement, generating result rows containing result columns. The SELECT statement must include the FOR BROWSE option.

- 2. Bind the result column values into program variables, one row at a time.
- 3. If appropriate, change the variables' values (possibly in response to user input).
- 4. If appropriate, construct and 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 reference the **timestamp** column. An appropriate WHERE clause can be constructed with the **dbqual** 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 (or primary key) and a **timestamp** column.
- The result columns to be used in the updates must be derived from tables that can be browsed and cannot be the result of compute columns, 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, browse mode requires two DBPROCESS structures: one for selecting the data and another for updating based on the selected data.

Building Applications

DB-Library applications can be built for the Microsoft® Windows NT® 4.0, Microsoft Windows® 95, Windows 98, and Microsoft MS-DOS® operating systems.

The following table shows the DB-Library include files used for all operating systems.

Include file	Contains
Sqlfront.h	Type definitions, error codes and severity levels,
	miscellaneous definitions.
Sqldb.h	Function prototypes for all DB-Library functions.
Changes to DB-Library for C

DB-Library for C version 7.0 has been changed and improved since version 4.2*x*, and your program can be revised to take advantage of these changes. However, no changes to your application or program source code are required. Existing applications should continue to run properly with DB-Library 7.0, and existing programs should build properly with DB-Library 7.0.

MSdblib3 Replaces W3dblib

The DB-Library files for Microsoft® Windows® have been renamed. The new files are named Msdblib3.dll and Msdblib3.lib. New DB-Library applications for Windows should use these files. The W3dblib.dll and W3dblib.lib files are still provided for backward compatibility with existing applications.

Header Files Sqlfront.h and Sqldb.h Changed

The DB-Library for C header files Sqlfront.h and Sqldb.h have been changed. They now use standard Microsoft® Win32® API portable data types. The new PDBPROCESS, PLOGINREC, and PDBCURSOR portable macros can be used instead of the platform-specific pointers.

Use	Instead of
PDBPROCESS	DBPROCESS *
	DBPROCESS NEAR *
	DBPROCESS FAR *
PLOGINREC	LOGINREC *
	LOGINREC NEAR *
	LOGINREC FAR *
PDBCURSOR	DBCURSOR *
	DBCURSOR NEAR *
	DBCURSOR FAR *

Also, the new PDBHANDLE macro indicates that either a PDBPROCESS, PLOGINREG, or PDBCURSOR may be valid.

Building Win32 DB-Library Applications

You can build DB-Library applications using the Microsoft® Win32® API, the 32-bit application programming interface for Microsoft Windows NT® 4.0, Microsoft Windows® 95, and Windows 98.

DB-Library Architecture

For the Microsoft® Win32®-based versions of the DB-Library dynamic-link library (DLL), a separate operating-system thread may be spawned for each connection that DB-Library makes with Microsoft SQL Server[™] (depending on the Net-Library in use). Each instance of the DB-Library DLL loaded by a calling process gets a private data area, while sharing code.

Win32-based DB-Library architecture differs from the implementation with the Microsoft Windows® operating system. In Windows, Msdblib3.dll maintains DB-Library connections as a linked list of connections in a single data segment. This architecture is required because Windows DLLs have a single data segment shared among all calling processes. This necessitates the initialization and clean up of the DB-Library DLL data structures through calls to the **dbinit** and **dbwinexit** functions. Because **dbinit** initializes private DB-Library structures and to maintain compatibility with future versions of DB-Library, you should call **dbinit** for all environments.

Libraries

The DB-Library functions for the Microsoft® Win32® API are stored in a dynamic-link library, Ntwdblib.dll. Net-Libraries for Microsoft Windows NT® 4.0, Microsoft Windows® 95, and Windows 98 are also implemented as DLLs. For example, Dbnmpntw.dll connects to Microsoft SQL ServerTM over named pipes. Set the PATH environment variable to include the directory in which the libraries reside.

The Ntwdblib.lib file is an import library containing function definitions that your applications use.

Include Files

You can append the path to the INCLUDE environment variable to include the directory where the DB-Library Sqlfront.h and Sqldb.h include files reside. Or you can use the /I compile line switch to point to the include file directory.

Because your Microsoft® Win32®-based applications must always include the Windows.h, Sqlfront.h, and Sqldb.h files, you don't need to define the DB-Library functions you use. These functions and their proper declarations are already defined in the include file. However, you must define an application's operating system before including the DB-Library include files. Include the following statements at the beginning of all DB-Library applications designed for the Win32 API:

#define DBNTWIN32
#include <windows.h>
#include <sqlfront.h>
#include <sqldb.h>

You can also define the operating system by using the /**D** compile line switch (for example, /**DDBNTWIN32** instead of a **#define** declaration).

Compiling and Linking

See the Microsoft® Visual C++® documentation for more information about compiling and linking a Microsoft Win32® application for Microsoft Windows NT® 4.0, Microsoft Windows® 95 or Windows 98. The DB-Library sample programs include sample build files.

Porting DB-Library Applications

All DB-Library functions are completely portable between the Microsoft® Windows® and Microsoft Windows NT® 4.0 platforms. You should not need to modify any of your DB-Library calls. You simply need to define the application's operating system at compile time (for example, **#define DBNTWIN32** for the Microsoft Win32® API).

Two DB-Library functions, **dbprhead** and **dbprrow**, not supported in Windows because they send output to STDIO, are supported in the Win32 API and can be used when developing applications for the CONSOLE subsystem. The **dbprhead** and **dbprrow** functions provide a convenient way to display the results of a query to the default output device.

If you have existing 16-bit DB-Library applications for Windows that you want to port to Win32 and run with Windows NT 4.0, Windows 95, or Windows 98, you can take two approaches to porting:

• Change your application source code so that it can be compiled for the Win32 or Windows platform.

If your DB-Library application is under development or is evolving to meet changing user needs, and you want to run the application with Windows and Windows NT 4.0, Windows 95, or Windows 98, make small changes to the source code to build 16-bit (Windows-based) and 32-bit (Win32-based) versions of your application from the same source.

• Port your application source code entirely to the Win32 platform.

If you decide to modify your application to run only as a 32-bit (Win32based) application (for example, you require Win32-specific features such as threads or preemptive multitasking), you can port your application to the Win32 API, removing redundant Windows calls and dependencies.

Porting to Maintain DB-Library Compatibility with Windows

You can port a 16-bit DB-Library application for the Microsoft® Windows® operating system to the Microsoft Win32® API for Microsoft Windows NT® 4.0, Windows 95 or Windows 98 operating systems while retaining compatibility with applications for Windows.

API Calls for Windows and C Run Time

In general, you can change your source code to easily build 16-bit or 32-bit versions. For an in-depth discussion of writing applications for the Win32 API that retain compatibility with the Windows operating system, see the *Microsoft Win32 API Programmer's Reference* (available separately).

Follow these guidelines when developing applications for the Win32 API:

- Be sure that your function definitions and prototypes use portable data types.
- The widening of handles to 32-bits means that the packing of handles and other values in *wParam* and *lParam* has changed. Verify whether or not you need to change the decoding of *wParam* and *lParam* in your application.

DB-Library Functions

All DB-Library functions are completely portable between operating systems. You simply need to define the target operating system at compile time.

Many DB-Library functions return values of type **int**. Note that the **int** data type is a 16-bit value in Windows and a 32-bit value in the Win32 API. Therefore, verify that your application uses the correct type of variable to receive return values from DB-Library functions that return an **int** data type. For example, a variable defined as the data type **short** receives a correct **int** value from DB-

Library in Windows but receives an incorrect value in the Win32 API.

Apart from verifying that variables receiving return values from DB-Library are of the correct type, no changes are needed to DB-Library functions.

Additional Porting Considerations

If you port a Windows-based application to the Microsoft® Windows NT® 4.0, Microsoft Windows® 95, or Windows 98 operating system and do not need to maintain compatibility with Windows, you can change your DB-Library code to remove redundant or unnecessary Windows functions.

Redundant Windows-specific DB-Library Functions

The **dbwinexit** function is Windows-specific and unnecessary in DB-Library for the Microsoft Win32® API.

Error/Message Handler Registration

The following changes apply to the registration of error and message handlers for the Win32 API:

• No need to export callback functions.

You don't need to export the DB-Library error and message handler callback functions in a .def file.

• No need to call **MakeProcInstance**.

You don't need to call **MakeProcInstance** to obtain a pointer to pass to the **dberrhandle** and **dbmsghandle** functions. Simply call **dberrhandle** and **dbmsghandle**, passing the function address. The following examples show the differences in calling the **dberrhandle** and **dbmsghandle** functions in Windows and in the Win32 API.

For Windows:

// Define variables
static FARPROC lpdbMsgHandler;
static FARPROC lpdbErrHandler;

// Get Procedure Instances

lpdbMsgHandler = MakeProcInstance((FARPROC)dbMsgHan
lpdbErrHandler = MakeProcInstance((FARPROC)dbErrHandle

// Install the instances into dblib
dbmsghandle(lpdbMessageHandler);
dberrhandle(lpdbErrorHandler);

For the Win32 API:

// Install the instances into dblib
dbmsghandle(dbMsgHandler);
dberrhandle(dbErrHandler);

Preemptive Multitasking (Win32) vs. Cooperative Multitasking (Windows)

To support asynchronous processing in the cooperative multitasking environment in Windows, each application must behave well and yield to the CPU at regular intervals. Therefore, when you process Microsoft SQL Server[™] queries in Windows, you need to use Windows timers or the **PeekMessage** function in combination with the DB-Library function calls for asynchronous processing (**dbsqlsend**, **dbdataready**, and **dbsqlok**).

Support for preemptive multitasking in the Win32 API simplifies the implementation of asynchronous query processing. The preemptive multitasking nature of the Windows NT, Windows 95 and Windows 98 operating systems ensures that other processes always obtain CPU cycles regardless of any processing that another application is doing. Within a single process, asynchronous processing can be implemented using Win32 threads or by using the DB-Library functions that support asynchronous processing in conjunction with the **PostMessage** Windows function. For more information about asynchronous processing, see <u>Taking Advantage of Win32 API Features in DB-Library Applications</u>.

Taking Advantage of Win32 API Features in DB-Library Applications

The Microsoft® Win32® API has many unique features.

Asynchronous Query Processing

Preemptive multitasking in the Microsoft® Win32® API makes it easy to implement asynchronous query processing. Use one of the following ways to implement asynchronous query processing, depending on whether you want your application to exhibit asynchronous behavior between processes or within a single process.

• Switch tasks between processes.

Use the standard **dbsqlexec** call to send a query to Microsoft SQL ServerTM. Although **dbsqlexec** is synchronous from the calling thread's perspective (**dbsqlexec** returns only when SQL Server processes the query and is ready to return results), the preemptive nature of the Win32 API allows other applications or threads of the same application to continue to work and process user input while the query is executing.

- Implement asynchronous processing within a single process:
 - Use threads.

The easiest way to implement asynchronous processing within a process is to use Win32 threads. You can spawn a thread that calls **dbsqlexec** and continue to do other work or continue to receive user input while the query is being processed.

• Single-thread approach.

If you require a single-thread process to implement asynchronous query processing, use the asynchronous DB-Library functions **dbsqlsend**, **dbdataready**, and **dbsqlok** in combination with the **PostMessage** function, as shown in this example:

```
case WM_SENDQUERY:
  dbsqlsend(dbproc);
  PostMessage(hWnd,WM_CHECKQUERY,0,0L);
  break;
```

```
case WM_CHECKQUERY:
    if (dbdataready(dbproc))
    {
        dbsqlok(dbproc);
        PostMessage(hWnd,WM_GETRESULTS,0,0L);
    }
    else
    {
        PostMessage(hWnd,WM_CHECKQUERY,0,0L);
    }
    break;
```

Memory Management in the Win32 API

The 32-bit linear virtual memory address space available to processes in the Microsoft® Win32® API makes memory management simpler and cleaner. (This applies to DB-Library programming as well as to all other programming for the Win32 API.)

Although the memory management functions (**GlobalAlloc**, **LocalAlloc**, and so on) in the Microsoft Windows® operating system are supported in the Win32 API, consider using the C run-time functions **malloc** and **free** to perform dynamic memory management when you develop applications to run with the Win32 API. In addition to the standard memory management functions (**Global** and **Local**, C run-time support), the Win32 API also provides heap and virtual memory management functions for applications that require specialized memory management.

Two specific areas of DB-Library programming can take advantage of the ability to address greater than 64 KB of data:

• Processing of **text** and **image** data.

There are many uses for Microsoft SQL Server[™] **text** and **image** data types. All require manipulation of **text** and **image** data, and the ability to transfer data between the application and SQL Server. The ability to address large chunks of memory can make this easier. Because it is possible to address up to 2 gigabytes (GB) of user memory in an application for the Win32 API, you can develop **text/image** handling functions that transfer **text/image** data in a single operation instead of transferring data chunks less than 64 KB. (Note, however, that waiting for 2 GB of data to transfer can take considerable time.)

• Row buffering.

In DB-Library for Microsoft MS-DOS® and Windows, you are constrained because DB-Library cannot handle more that 64 KB of data at a time, including memory allocated for row buffering. This value is further reduced by overhead taken up by data structures, variables, and so on. In the Win32 API, it is possible to use DBSETOPT(DBBUFFER,...) to enable row buffering for data sets greater than 64 KB.

Threads in the Win32 API

The multithreaded process structure of the Microsoft® Win32® API opens up powerful and elegant solutions to high-end applications. Each thread within a process has its own stack and computer state. On multiprocessor systems with the Microsoft Windows NT® 4.0 operating system, multiple threads can execute at the same time but on different processors. Applications that need to perform intensive processing, I/O, and so on, can do this in the background using threads and still offer a responsive user interface.

Using DB-Library in a Threaded Application

Here are a few tips to be aware of when creating a threaded DB-Library application:

- Make sure you call the **dbinit** and **dbexit** functions only one time on your application.
- Make sure to use per process error and message handling.
- If you use global error and message handlers, you may need to synchronize access in your error and message handler routines.
- Make sure to call **dbsetmaxprocs** if you are going to have more than 25 connections open.
- Make sure you are compiling/linking with the multithreaded C/C++ run-time libraries. By default, a Microsoft® Visual C/C++® project is linked with the single threaded run-time libraries.
- If you use the same LOGINREC to open multiple connections, access to the LOGINREC must be serialized.

See Also

<u>dbprocerrhandle</u>

<u>dbprocmsghandle</u>

Reentrancy and DB-Library

In applications where each thread uses a separate DBPROCESS connection, you don't need to serialize the DB-Library calls. However, you must use **dbprocerrhandle** (instead of **dberrhandle**) and **dbprocmsghandle** (instead of **dbmsghandle**) to establish connection-specific (instead of global) error and message handlers for your application. You should pass a LOGINREC to **dbprocerrhandle** and **dbprocmsghandle** before calling **dbopen** using that LOGINREC. If you follow these steps, DB-Library is completely reentrant and thread-safe.

DB-Library functions and routines that access a shared DBPROCESS are not reentrant across multiple threads. Therefore, be sure that you serialize all DB-Library calls that access the same DBPROCESS in multithreaded applications you develop.

DB-Library for C Reference

DB-Library for C is an application programming interface (API) consisting of C functions and macros that allow an application to interact with Microsoft® SQL ServerTM 2000. Included are functions that send Transact-SQL statements to SQL Server and functions that process the results of those statements.

Core Functions

DB-Library for C version 7.0 supports several core functions.

dbadata

Returns a pointer to the data for a compute column.

Syntax

LPCBYTE dbadata (PDBPROCESS dbproc, INT computeid, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation/ Microsoft® SQL ServerTM 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

Is the ID that identifies 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 **dbnextrow** or **dbgetrow**.

column

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

Returns

A BYTE pointer to the data for a particular column in a compute. A null BYTE pointer is returned if there is no such column or compute or if the data has a null value. The data space pointed to is allocated and freed by DB-Library. Be careful not to overwrite the space.

Remarks
After each call to **dbnextrow** that returns a value greater than 0, use **dbadata** to obtain a pointer to the data for a particular column in a compute. The data is not null-terminated. Use **dbadlen** to get the length of the data.

When a column of integer data is summed or averaged, SQL Server always returns a four-byte integer, regardless of the size of the column. Therefore, be sure that the variable that is to contain the result from such a compute is declared as DBINT.

Examples

The following program fragment shows how to use **dbadata**:

```
DBPROCESS *dbproc;
int rowinfo;
DBINT sum;
```

```
// First, put the commands into the command buffer.
dbcmd(dbproc, "SELECT fileid, size FROM sysdevices");
dbcmd(dbproc, " ORDER BY fileid ");
dbcmd(dbproc, " COMPUTE SUM(size) BY fileid ");
// Send the commands to SQL Server and start execution.
dbsqlexec(dbproc);
// Process the command.
dbresults(dbproc);
// Examine the results of the COMPUTE clause.
while((rowinfo = dbnextrow(dbproc)) != NO_MORE_ROWS)
{
 if (rowinfo == REG ROW)
   printf("Regular row returned.\n");
 else
  {
   // This row is the result of a COMPUTE clause,
   // and "rowinfo" is the computeid of this COMPUTE
   // clause.
```

```
sum = *(DBINT *)(dbadata(dbproc, rowinfo, 1));
printf("sum = %ld\n", sum);
}
```

The **dbaltbind** function automatically binds data to your program variables. It is somewhat easier to use than **dbadata** and **dbadlen**, but is less efficient because it copies the data into your variable.

See Also dbadlen dbgetrow dbaltbind dbnextrow dbaltlen dbnumalts dbalttype

dbadlen

Returns the actual length of the data for a compute column.

Syntax

DBINT dbadlen (PDBPROCESS dbproc, INT computeid, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

Is the ID that identifies 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 **dbnextrow** or **dbgetrow**.

column

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

Returns

The length, in bytes, of the data for 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

This **dbadlen** returns the length of the data for a compute column. You can get a pointer to the actual data by using **dbadata**. Calling **dbadata** after **dbnextrow** or **dbgetrow** returns a *computeid*.

Examples

The following program fragment shows how to use **dbadlen**:

DBPROCESS *dbproc; char biggest_name[MAXNAME+1]; DBINT namelen; STATUS rowinfo;

```
// Put the command into the command buffer.
dbcmd(dbproc, "SELECT name FROM sysobjects");
dbcmd(dbproc, " ORDER BY name");
dbcmd(dbproc, " COMPUTE MAX(name)");
```

// Send the command to SQL Server and start execution.
dbsqlexec(dbproc);

```
// Process the command.
dbresults(dbproc);
```

```
// Examine each row returned by the command.
while ((rowinfo = dbnextrow(dbproc)) != NO_MORE_ROWS)
{
    if (rowinfo == REG_ROW)
        printf("Regular row returned.\n");
    else
    {
        // This row is the result of a COMPUTE clause,
        // and "rowinfo" is the computeid of this COMPUTE
        // clause.
```

```
namelen = dbadlen(dbproc, rowinfo, 1);
strncpy(biggest_name,(char *)dbadata(dbproc, rowinfo, 1),
  (int)namelen);
// Data pointed to by dbadata() is not null-terminated.
biggest_name[namelen] = '\0';
```

```
printf("biggest name = %s\n", biggest_name);
}
```

See Also

<u>dbadata</u>

<u>dbgetrow</u>

<u>dbaltlen</u>

<u>dbnextrow</u>

<u>dbalttype</u>

<u>dbnumalts</u>

dbaltbind

Binds a *compute column* (column of results from a COMPUTE clause) to a program variable.

Syntax

RETCODE dbaltbind (PDBPROCESS dbproc, INT computeid, INT column, INT vartype, DBINT varlen, LPCBYTE varaddr);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

Is the ID of the COMPUTE clause to which the **dbaltbind** function refers. Because a SELECT statement can have more than one COMPUTE clause, the *computeid* is necessary to distinguish between them. The *computeid* is returned by **dbnextrow** or **dbgetrow**.

column

Is the column number of the compute data that is to be copied to a program variable. The first column is number 1.

vartype

Is a description of the binding's data type. It corresponds to the data type of the program variable that receives the copy of the data from the

DBPROCESS.

The **dbaltbind** function supports a wide range of type conversions, so the binding's data type can be different from the type returned by the SQL query. For instance, a SQLMONEY result can be bound to a DBFLT8 program variable, using FLT8BIND, and the appropriate data conversion happens automatically.

For more information about a list of the data conversions provided by DB-Library, see <u>dbwillconvert</u>. For more information about a list of the type definitions used by DB-Library, see <u>DB-Library for C Data types</u>. The following table lists the legal *vartypes* recognized by **dbaltbind** and the program variable and SQL Server data types to which each refers.

	Program variable data	SQL Server data
vartype	type	type
CHARBIND	DBCHAR	SQLCHAR
STRINGBIND	DBCHAR	SQLCHAR
NTBSTRINGBIND	DBCHAR	SQLCHAR
VARYCHARBIND	DBVARYCHAR	SQLCHAR
BINARYBIND	DBBINARY	SQLBINARY
VARYBINBIND	DBVARYBIN	SQLBINARY
TINYBIND	DBTINYINT	SQLINT1
SMALLBIND	DBSMALLINT	SQLINT2
INTBIND	DBINT	SQLINT4
FLT8BIND	DBFLT8	SQLFLT8
BITBIND	DBBIT	SQLBIT
DATETIMEBIND	DBDATETIME	SQLDATETIME
MONEYBIND	DBMONEY	SQLMONEY
SMALLMONEYBIND	DBMONEY4	SQLMONEY4
SMALLDATETIBIND	DBDATETIM4	SQLDATETIM4
FLT4BIND	DBFLT4	SQLFLT4
DECIMALBIND	DBDECIMAL	SQLDECIMAL
NUMERICBIND	DBNUMERIC	SQLNUMERIC
SRCDECIMALBIND	DBDECIMAL	SQLDECIMAL
SRCNUMERICBIND	DBNUMERIC	SQLNUMERIC

Because SQLTEXT and SQLIMAGE data are never returned through a *compute row* (a row of results generated by a COMPUTE clause), these data types are not included in the preceding table. The SQL Server data type is listed for your reference. The *vartype* you specify does not necessarily have to correspond to a particular SQL Server data type because, as mentioned earlier, **dbaltbind** converts SQL Server data into the specified *vartype*.

The following table lists the four representations for character data. They differ according to whether the data is padded with blanks or is null-terminated:

vartype	Program data type	Padding	Terminator
CHARBIND	DBCHAR	blanks	none
STRINGBIND	DBCHAR	blanks	\0
NTBSTRINGBIND	DBCHAR	none	\0
VARYCHARBIND	DBVARYCHAR	none	none

Note that "\0" is the null terminator character. Similarly, binary data can be stored in two different ways:

vartype	Program data type	Padding
BINARYBIND	DBBINARY	nulls
VARYBINBIND	DBVARBINARY	none

When a column of integer data is summed or averaged, SQL Server always returns a four-byte integer, regardless of the size of the column. Therefore, be sure that the variable that is to contain the result from such a compute is declared as DBINT and that the *vartype* of the binding is INTBIND.

When the source column specified by the *column* parameter has a type of SQLDECIMAL or SQLNUMERIC, you can keep the same precision and scale in your bound C variable by using SRCDECIMALBIND or SRCNUMERICBIND.

varlen

Is the length of the program variable in bytes. For fixed-length *vartypes*, such as MONEYBIND or FLT8BIND, this length is ignored. For character and binary types, *varlen* must describe the total length of the available destination buffer space, including any space required for special terminating bytes, such as a null terminator. If *varlen* is 0, the total number of bytes available is copied into the program variable. (For **char** and **binary** SQL Server data, the total number of bytes available is equal to the defined length of the database column, including any blank padding. For **varchar** and **varbinary** data, the total number of bytes available is equal to the actual data contained in the column.) Therefore, if you are sure that your program variable is large enough to handle the results, set *varlen* to 0. The *varlen* is ignored for VARYCHARBIND and VARYBINBIND data.

varaddr

Is the address of the program variable to which the data is copied. Calling **dbaltbind** with a NULL *var* address parameter breaks previously set bindings.

When binding using DECIMALBIND or NUMERICBIND, the *varaddr* parameter must be a pointer to a DBNUMERIC or DBDECIMAL C variable, respectively, with the *precision* and *scale* fields of the structure already set to the desired values. You can use DEFAULTPRECISION to specify a default precision and DEFAULTSCALE to specify a default scale.

Returns

SUCCEED or FAIL. The **dbaltbind** function returns FAIL if *vartype* isn't compatible with the SQL Server data type being returned, or if *varaddr* is NULL.

Remarks

This function directs DB-Library to copy compute column data returned by SQL Server into a program variable. When each new row containing computed data is read by **dbnextrow** or **dbgetrow**, the data from the designated *column* in that compute row is copied into the program variable with the address *varaddr*. There must be a separate **dbaltbind** call for each compute column to be copied. It is not necessary to bind every compute column to a program variable.

SQL Server can return two types of rows: regular rows containing data from columns designated by a SELECT statement's select list, and compute rows resulting from the COMPUTE clause. The **dbaltbind** function binds data from compute rows. Use **dbbind** for binding data from regular rows.

The calls to **dbaltbind** must be made after a call to **dbresults** and before the first call to **dbnextrow**.

Using **dbaltbind** causes some overhead because it always copies the row data into the designated program variable. To avoid this copying, the returned data can be accessed more directly with **dbadlen** and **dbadata**.

Because null values can be returned from SQL Server, there is a set of default values, one for each data type that is substituted when binding null values. You can explicitly set your own values to be substituted for the default null values with the **dbsetnull** function. (For more information about a list of the default substitution values, see <u>dbsetnull</u>.)

Examples

This example shows the typical sequence of calls:

```
DBCHAR name[20];
DBINT namecount;
```

// Read the query into the command buffer. dbcmd(dbproc, "SELECT name FROM employee COMPUTE COUN" // Send the query to SQL Server. dbsqlexec(dbproc); // Get ready to process the results of the query. dbresults(dbproc); // Bind the regular row data - name. dbbind(dbproc, 1, STRINGBIND, (DBINT) 0, name); // Bind the compute column data - count of name. dbaltbind(dbproc, 1, 1, INTBIND, (DBINT) 0, (BYTE *) &namecount // Now process each row.

```
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
    //C-code to print or process row data
}
```

See Also

DB-Library for C Data Types

<u>dbgetrow</u>

<u>dbadata</u>

<u>dbnextrow</u>

<u>dbadlen</u>

<u>dbresults</u>

<u>dbanullbind</u>

<u>dbsetnull</u>

<u>dbbind</u>

<u>dbwillconvert</u>

<u>dbconvert</u>

dbaltcolid

Returns the operand column ID for a compute column.

Syntax

INT dbaltcolid (PDBPROCESS dbproc, INT computeid, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

Is the ID that identifies 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 **dbnextrow** or **dbgetrow**.

column

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

Returns

The column ID that the aggregate in the compute applies to. The first column is number 1. If either *computeid* or *column* is invalid, -1 is returned. Call this function after **dbnextrow** or **dbgetrow** returns a *computeid*.

Examples

When issued after the following SELECT statement, **dbaltcolid(dbproc, 1, 1)** returns 2, because the COMPUTE COUNT clause in the SELECT statement 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

<u>dbadata</u>

<u>dbnextrow</u>

<u>dbadlen</u>

<u>dbnumalts</u>

<u>dbaltlen</u>

<u>dbprtype</u>

<u>dbgetrow</u>

dbaltlen

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

Syntax

DBINT dbaltlen (PDBPROCESS dbproc, INT computeid, INT column);

Arguments

dbproc

The DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL ServerTM 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

The ID that identifies 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 **dbnextrow** or **dbgetrow**.

column

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

Returns

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

Remarks

dbaltlen returns the maximum length, in bytes, for a particular column in a

compute row. For variable-length data, this is not necessarily the actual length of the data, but rather the maximum length. Calling **dbaltlen** only after **dbnextrow** or **dbgetrow** returns a *computeid*. For the actual data length, use **dbadlen**.

Examples

After the following SELECT statement, calling **dbaltlen(dbproc, 1, 1)** returns 4, because counts are of SQLINT4 type, which is 4 bytes long:

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

See Also

<u>dbadata</u>

<u>dbgetrow</u>

<u>dbadlen</u>

<u>dbnextrow</u>

<u>dbalttype</u>

<u>dbnumalts</u>

dbaltop

Returns the type of aggregate function for a compute column.

Syntax

INT dbaltop (PDBPROCESS dbproc, INT computeid, INT column);

Arguments

dbproc

The DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL ServerTM 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

The ID that identifies 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 **dbnextrow** or **dbgetrow**.

column

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

Returns

The type of aggregate operator for the particular column in the compute. The types are defined as follows:

Туре	Aggregate operator
SQLAOPSUM	SUM
SQLAOPAVG	AVG

SQLAOPCNT	COUNT
SQLAOPMIN	MIN
SQLAOPMAX	MAX

Call **dbaltop** only after **dbnextrow** or **dbgetrow** returns a *computeid*.

If *computeid* or *column* is not valid, -1 is returned.

Examples

After the following SELECT statement, calling **dbaltop(dbproc, 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 **dbprtype**.

See Also

<u>dbadata</u>

<u>dbnextrow</u>

<u>dbadlen</u>

<u>dbnumalts</u>

<u>dbaltlen</u>

<u>dbprtype</u>

<u>dbgetrow</u>

dbalttype

Returns the data type for a compute column.

Syntax

INT dbalttype (PDBPROCESS dbproc, INT computeid, INT column);

Arguments

dbproc

The DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL ServerTM 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

The ID that identifies 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 **dbnextrow** or **dbgetrow**.

column

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

Returns

The type of the data for the particular column in the compute. The types are defined as follows:

Column data type	Returned constant
Char	SQLCHAR
Varchar	SQLCHAR

Binary	SQLBINARY
Varbinary	SQLBINARY
Tinyint	SQLINT1
Smallint	SQLINT2
Int	SQLINT4
Real	SQLFLT4
Float	SQLFLT8
Smallmoney	SQLMONEY4
Money	SQLMONEY
Decimal	SQLDECIMAL
Numeric	SQLNUMERIC
Smalldatetime	SQLDATETIM4
Datetime	SQLDATETIME
Image	SQLIMAGE
Text	SQLTEXT

Call **dbalttype** only after **dbnextrow** or **dbgetrow** returns a *computeid*.

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

Examples

In the following SELECT statements, calling **dbalttype(dbproc, 1, 1)** returns the type for SQLINT4 because counts are of SQLINT4 type.

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

To convert the type to a readable string, use **dbprtype**.

See Also

DB-Library for C Data Types

<u>dbgetrow</u>

<u>dbadata</u>

<u>dbnextrow</u>

<u>dbadlen</u>

<u>dbnumalts</u>

<u>dbaltlen</u>

<u>dbprtype</u>

dbaltutype

Returns the user-defined data type for a compute column.

Syntax

DBINT dbaltutype (PDBPROCESS dbproc, INT computeid, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

Is the ID that identifies 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 **dbnextrow** or **dbgetrow**.

column

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

Returns

The user-defined data type of the specified compute column on success and -1 on error.

Remarks

This function is defined as type DBINT, because both the DB-Library data type

DBINT and user-defined data types are 32 bits long. Call **dbaltutype** only after **dbnextrow** or **dbgetrow** returns a *computeid*.

See Also

<u>dbalttype</u>

<u>dbcolutype</u>

dbanullbind

Associates an indicator variable with a compute-row column.

Syntax

RETCODE dbanullbind (PDBPROCESS dbproc, INT computeid, INT column, LPCDBINT indicator);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

Is the ID that identifies 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 **dbnextrow** or **dbgetrow**. A value of 1 is the *computeid* corresponding to the first compute clause in a SELECT statement.

column

Is the number of the column to be associated with the indicator variable.

indicator

Is a pointer to the indicator variable.

Note The *indicator* parameter is just the pointer to the indicator variable. The variable itself is set.

Returns

SUCCEED or FAIL. The **dbanullbind** function returns FAIL if *computeid* or *column* is invalid.

Remarks

The indicator variable reveals whether a particular compute-row column has been converted and copied to a program variable successfully or unsuccessfully, or whether it is NULL. Call **dbanullbind** only after **dbnextrow** or **dbgetrow** returns a *computeid*.

The indicator variable is set when compute rows are processed through **dbnextrow**. The possible values are:

- -1 if the column is NULL.
- The full length of the column's data. This value is in bytes if the column was bound to a program variable through **dbaltbind**; the binding did not specify any data conversions: and the bound data was truncated because the

any data conversions; and the bound data was truncated because the program variable was too small to hold the column's data.

• 0 if the column was successfully bound and copied to a program variable.

Note Detection of character string truncation is implemented only for CHARBIND and VARYCHARBIND.

See Also

<u>dbadata</u>

<u>dbnextrow</u>

<u>dbadlen</u>

<u>dbnullbind</u>

<u>dbaltbind</u>

dbbind

Binds a regular result column (a column of results from a SELECT statement's select list) to a program variable.

Syntax

RETCODE dbbind (PDBPROCESS dbproc, INT column, INT vartype, DBINT varlen, LPBYTE varaddr);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the column number of the row data to be copied to a program variable. The first column is number 1.

vartype

Is a description of the binding's data type. It corresponds to the data type of the program variable that receives the copy of the data from the DBPROCESS. The **dbbind** function supports a wide range of type conversions, so *vartype* can be different from the type returned by the SQL query. For instance, a SQLMONEY result can be bound to a DBFLT8 program variable, using FLT8BIND, and the appropriate data conversion happens automatically. For more information about a list of the data conversions provided by DB-Library, see <u>dbwillconvert</u>. For more information about a list of the type <u>DB-Library</u>, see <u>DB-Library</u>, s

Library for C Data Types.

The following table lists legal *vartypes* recognized by **dbbind** and the program variable and SQL Server type tokens that each refers to.

		SQL Server data type
vartype	varaddr	of <i>column</i>
CHARBIND	DBCHAR	SQLCHAR, SQLVARCHAR,
		or SQLTEXT
STRINGBIND	DBCHAR	SQLCHAR, SQLVARCHAR,
		or SQLTEXT
NTBSTRINGBIND	DBCHAR	SQLCHAR, SQLVARCHAR,
		or SQLTEXT
VARYCHARBIND	DBVARYCHAR	SQLCHAR, SQLVARCHAR,
		or SQLTEXT
BINARYBIND	DBBINARY	SQLBINARY,
		SQLVARBINARY, or
		SQLIMAGE
VARYBINBIND	DBVARYBIN	SQLBINARY,
		SQLVARBINARY, or
		SQLIMAGE
TINYBIND	DBTINYINT	SQLINT1 or SQLINTN
SMALLBIND	DBSMALLINT	SQLINT2 or SQLINTN
INTBIND	DBINT	SQLINT4 or SQLINTN
FLT4BIND	DBFLT4	SQLFLT4 or SQLFLTN
FLT8BIND	DBFLT8	SQLFLT8 or SQLFLTN
BITBIND	DBBIT	SQLBIT
SMALLMONEYBIND	DBMONEY4	SQLMONEY4 or
		SQLMONEYN
MONEYBIND	DBMONEY	SQLMONEY or
		SQLMONEYN
DECIMALBIND	DBDECIMAL	SQLDECIMAL
NUMERICBIND	DBNUMERIC	SQLNUMERIC
SRCDECIMALBIND	DBDECIMAL	SQLDECIMAL
SRCNUMERICBIND	DBNUMERIC	SQLNUMERIC

SMALLDATETIBIND	DBDATETIM4	SQLDATETIM4 or SQLDATETIMN
DATETIMEBIND	DBDATETIME	SQLDATETIME or SQLDATETIMN

Note that the SQL Server type in the preceding table is listed merely for your information. The *vartype* you specify does not necessarily have to correspond to a particular SQL Server data type because **dbbind** converts SQL Server data into the specified *vartype*.

The following table lists the four representations for character and text data. They differ according to whether the data is blank-padded or null-terminated.

vartype	varaddr	Padding	Terminator
CHARBIND	DBCHAR	blanks	none
STRINGBIND	DBCHAR	blanks	\0
NTBSTRINGBIND	DBCHAR	none	\0
VARYCHARBIND	DBVARYCHAR	none	none

Note that "\0" is the null terminator character. Similarly, binary and image data can be stored in two different ways.

vartype	varaddr	Padding
BINARYBIND	DBBINARY	nulls
VARYBINBIND	DBVARBINARY	none

When the source column specified by the *column* parameter has a type of SQLDECIMAL or SQLNUMERIC, you can keep the same precision and scale in your bound C variable by using SRCDECIMALBIND or SRCNUMERICBIND.

varlen

Is the length of the *varaddr* program variable in bytes. For fixed-length *vartypes*, such as MONEYBIND or FLT8BIND, this length is ignored. For
character, text, binary, and image types, *varlen* must describe the total length of the available destination buffer space, including any space that can be required for special terminating bytes, such as a null terminator. If *varlen* is 0, the number of bytes available is copied into the program variable. (For **char** and **binary** SQL Server data, the number of bytes available is equal to the defined length of the database column, including any blank padding. For **varchar**, **varbinary**, **text**, and **image** data, the number of bytes available is equal to the actual data contained in the column.) Therefore, if you are sure that your program variable is large enough to handle the results, you can set *varlen* to 0.

In some cases, DB-Library issues a message indicating that data conversion resulted in an overflow. This is usually caused by a *varlen* specification being too small for the data being received from SQL Server. For example, if *varlen* is set to 5, *vartype* is set to VARYCHARBIND, and the SQL Server column being bound is of type VARCHAR with a length of 20. When the bind occurs (using **dbnextrow**), the overflow message is issued. Note however that five bytes of data will be bound. Other types of binds also can cause the overflow message to be issued. For information about data type conversions, see <u>dbconvert</u>.

varaddr

Is the address of the program variable to which the data is copied. Calling **dbbind** with a NULL *varaddr* parameter breaks a previously set binding.

When binding using DECIMALBIND or NUMERICBIND, the *varaddr* parameter must be a pointer to a DBNUMERIC or DBDECIMAL C variable, respectively, with the *precision* and *scale* fields of the structure already set to the desired values. You can use DEFAULTPRECISION to specify a default precision and DEFAULTSCALE to specify a default scale.

Returns

SUCCEED or FAIL. The **dbbind** function returns FAIL if the column number given isn't valid, if the *vartype* isn't compatible with the SQL Server data type being returned, or if *varaddr* is NULL.

Remarks

Data comes back from SQL Server one row at a time. This function directs DB-Library to copy the data for a regular column (designated in a SELECT statement's select list) into a program variable. When each new row containing regular (not compute) data is read using **dbnextrow** or **dbgetrow**, the data from the designated *column* in that row is copied into the program variable with the address *varaddr*. There must be a separate **dbbind** call for each regular column to be copied. It is not necessary to bind every column to a program variable. A result column can be bound to only one program variable.

SQL Server can return two types of rows: regular rows and compute rows resulting from the COMPUTE clause of a SELECT statement. The **dbbind** function binds data from regular rows. Use **dbaltbind** for binding data from compute rows.

Calls to **dbbind** must be made after a call to **dbresults** and before the first call to **dbnextrow**.

Using **dbbind** causes some overhead because it copies the row data into the designated program variable. To avoid this copying, the returned data can be accessed more directly with **dbdatlen** and **dbdata**.

Because null values can be returned from SQL Server, there is a set of default values, one for each data type that will be substituted when binding null values. You can explicitly set your own values to be substituted for the default null value with the **dbsetnull** function. (For more information about a list of the default substitution values, see <u>dbsetnull</u>.)

For the Microsoft Windows® operating system, DB-Library retrieves information about date, time, numeric, and currency formatting from the Sqlcommn.loc file. The location of Sqlcommn.loc is pointed to by the **SQLLocalizationFile** key in the Windows initialization file (Win.ini) under the [**SQLSERVER**] application heading. For example:

[SQLSERVER] SQLLocalizationFile=C:\SQL60\BIN\SQLCOMMN.LOC

For the Microsoft Windows NT® 4.0 operating system, you set the date, time, numeric, and currency formatting using the International application in Control Panel. Use the SQL Server Client Network Utility **Use International Settings** option to activate this for DB-Library.

Examples

This example shows the typical sequence of calls:

```
DBINT xvariable;
DBCHAR yvariable[10];
```

```
// Read the query into the command buffer.
dbcmd(dbproc, "SELECT x = 100, y = 'hello''');
// Send the query to SQL Server.
dbsqlexec(dbproc);
// Get ready to process the results of the query.
dbresults(dbproc);
// Bind column data to program variables.
dbbind(dbproc, 1, INTBIND, (DBINT) 0, (BYTE *) &xvariable);
dbbind(dbproc, 2, STRINGBIND, (DBINT) 0, yvariable);
```

```
// Now process each row.
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
    // C-code to print or process row data
}
```

See Also

DB-Library for C Data Types

dbgetrow

<u>dbaltbind</u>

<u>dbnextrow</u>

<u>dbanullbind</u>

<u>dbresults</u>

<u>dbconvert</u>

dbsetnull dbdata dbwillconvert dbdatlen

dbbylist

Returns the bylist for a compute row.

Syntax

LPCBYTE dbbylist (PDBPROCESS dbproc, INT computeid, LPINT size);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

Is the ID that identifies the COMPUTE clause. A SELECT statement can have multiple COMPUTE clauses, which can have varying numbers of aggregate operators and targets. The *computeid* is returned by **dbnextrow** or **dbgetrow**.

size

Is a pointer to an integer which **dbbylist** sets to the number of elements in the bylist.

Returns

A pointer to an array of BYTES containing the numbers of the columns that compose the bylist for the specified compute. The names of the columns are available by calling **dbcolname**. If the *computeid* is out of range, NULL is returned.

The size of the array is returned in the *size* parameter. A *size* of 0 indicates that either this particular compute has no bylist or the *computeid* is out of range.

Remarks

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

Call this function after **dbresults** has returned SUCCEED.

Examples

Assume the following command has been executed:

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

The call **dbbylist(dbproc, 1, &size)** sets *size* to 2 because there are two items in the bylist. It returns a pointer to an array of 2 BYTES, which contain the values 1 and 2, indicating that the bylist is composed of columns 1 and 2 from the select list.

See Also

<u>dbadata</u>

<u>dbcolname</u>

<u>dbadlen</u>

<u>dbgetrow</u>

<u>dbaltlen</u>

<u>dbnextrow</u>

<u>dbalttype</u>

<u>dbresults</u>

dbcancel

Cancels the current command batch and flushes any pending results.

Syntax

RETCODE dbcancel (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL.

Remarks

You can call this function after calling **dbsqlexec**, **dbsqlsend**, **dbsqlok**, **dbresults**, or **dbnextrow** to cancel the execution of the current command batch that SQL Server is processing and eliminate any pending results. When **dbcancel** is called, SQL Server is interrupted and stops executing the command batch associated with the *dbproc*. Any pending results are read and discarded. Note that **dbcancel** cancels all the commands in the current command batch. Do not use it to cancel only the current command in a multiple-command batch; use **dbcanquery** instead.

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

See Also

dbcanquery

<u>dbsqlexec</u>

<u>dbnextrow</u>

<u>dbsqlok</u>

<u>dbresults</u>

<u>dbsqlsend</u>

dbcanquery

Cancels any rows pending from the most recently executed query.

Syntax

RETCODE dbcanquery (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL.

Remarks

This function is an efficient way to throw away any unread rows that result from the most recently executed SQL query. Calling **dbcanquery** is similar to calling **dbnextrow** until it returns NO_MORE_ROWS. Binds are broken before rows are read.

The **dbresults** function must return SUCCEED before an application can call **dbcanquery**.

If you want to ignore all of the results from all of the commands in the current command batch, call **dbcancel**.

See Also dbcancel dbresults <u>dbnextrow</u>

<u>dbsqlexec</u>

dbchange

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

Syntax

LPCSTR dbchange (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

A pointer to the null-terminated name of the new database, if any. If the database has not changed, NULL is returned.

Remarks

The **dbchange** function informs the program of a switch from one database to another by catching any instance of the Transact-SQL USE statement.

When connected to SQL Server version 4.2, a USE statement does not take effect until the end of the batch. The **dbchange** 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 **dbchange** when **dbresults** returns NO_MORE_RESULTS at the end of each command batch.

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

Alternatively, you can always get the name of the current database by calling

dbname.

See Also

<u>dbname</u>

<u>dbsqlsend</u>

<u>dbresults</u>

<u>dbuse</u>

<u>dbsqlexec</u>

dbclose

Closes and frees a single DBPROCESS structure.

Syntax

RETCODE dbclose (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL

Remarks

The **dbclose** function is the inverse of **dbopen**. It cleans up any activity associated with a single DBPROCESS structure and frees the memory. It also closes the corresponding network connection and any open cursors.

To close all open DBPROCESS structures, use **dbexit**.

Calling **dbclose** with a parameter not returned by **dbopen** causes an error.

See Also

<u>dbexit</u>

<u>dbopen</u>

dbclrbuf

Clears rows from the row buffer.

Syntax

void dbclrbuf (
PDBPROCESS dbproc,
DBINT n);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

n

Is the number of rows you want cleared from the row buffer. If *n* is equal to or greater than the number of rows in the buffer, all but the newest row are removed. If *n* is less than 1, the call is ignored. Rows are cleared on a first-in/first-out basis.

Remarks

You can turn row buffering on by calling **dbsetopt** (*dbproc*, DBBUFFER, *n*), where *n* is the number of rows you would like DB-Library to buffer. DB-Library for C can buffer rows as they are returned from SQL Server. If buffering is on, you can refer to buffered rows that have been read from SQL Server using **dbgetrow**.

The row buffer can become full if SQL Server returns more than the *n* rows that you said you wanted buffered. The row buffer is full when **dbnextrow** returns BUF_FULL. After the row buffer is full, **dbnextrow** continues to return BUF_FULL until at least one row is freed by calling **dbclrbuf**. The **dbclrbuf**

function frees the oldest n rows in the buffer.

See Also

<u>dbgetrow</u>

<u>dbnextrow</u>

DB-Library Options

<u>dbsetopt</u>

dbclropt

Clears an option set by **dbsetopt**.

Syntax

RETCODE dbclropt (PDBPROCESS dbproc, INT option, LPCSTR param);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL ServerTM 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server. If *dbproc* is NULL, the option is set for all active and future DBPROCESS structures.

option

Is the option to be turned off.

param

Is the option parameter. Certain options take parameters. If an option does not take a parameter, *param* is ignored. If an option does take a parameter, *param* is ignored for all options except DBOFFSET and DBSTAT.

The DBOFFSET and DBSTAT options can have several settings, each with a different parameter. In these cases, **dbclropt** needs a valid *param* to determine which option parameter to clear.

For more information about a list of options that take parameters, see <u>DB-Library Options</u>.

If an invalid parameter is specified for one of the SQL Server options, it will not be discovered until the command is sent to SQL Server. The **dbsqlexec**

or **dbsqlsend** calls will fail and DB-Library will invoke the user-installed message handler. If an invalid parameter is specified for one of the DB-Library options (DBBUFFER or DBTEXTLIMIT), **dbclropt** itself will fail.

Returns

SUCCEED or FAIL.

Remarks

This function clears SQL Server and DB-Library options that have been set with **dbsetopt**. Although SQL Server options can be set and cleared directly through Transact-SQL, use **dbsetopt** and **dbclropt** in an application to set and clear options. This provides a uniform interface for setting both SQL Server and DB-Library options. It also allows the application to use **dbisopt** to check the status of an option.

The **dbclropt** function does not immediately clear the options specified. With the exception of DBBUFFER and DBNOAUTOFREE, options are not cleared until the command buffer is sent to SQL Server (by invoking the **dbsqlexec** function). An additional result is returned by using **dbresults**. For information about results returned, see **dbresults**.

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 **dbsqlexec** 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 complete list of options, see <u>DB-Library Options</u>.

See Also

<u>dbisopt</u> <u>DB-Library Options</u> <u>dbsetopt</u>

dbcmd

Adds text to the DBPROCESS command buffer.

Syntax

RETCODE dbcmd (PDBPROCESS dbproc, LPCSTR cmdstring);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

cmdstring

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

Returns

SUCCEED or FAIL.

Remarks

This function adds text to the command buffer in the DBPROCESS structure. It adds to the existing command buffer; it doesn't delete or overwrite the current contents except after the buffer has been sent to SQL Server. The user can call **dbcmd** repeatedly. Note that sequential calls are concatenated; the application must include any necessary blanks between the end of one line and the beginning of the next.

After a call to **dbsqlexec** or **dbsqlsend**, the first call to either **dbcmd** or **dbfcmd** automatically clears the command buffer before the new text is entered. If this situation is undesirable, set the DBNOAUTOFREE option. When

DBNOAUTOFREE is set, the command buffer is cleared only by a call to **dbfreebuf**.

The **dbfcmd** function is a related function. Unlike **dbcmd**, **dbfcmd** takes additional parameters and interprets *cmdstring* as a format string that is passed to **sprintf** along with any additional parameters.

Examples

This example shows how to use **dbcmd** to build up a multiple-line SQL statement:

```
DBPROCESS *dbproc;
```

```
dbcmd(dbproc, "SELECT name FROM sysobjects");
dbcmd(dbproc, "WHERE id < 5");
dbcmd(dbproc, "AND type='S'");
```

Note the leading spaces in the second and third calls to **dbcmd**. Leading spaces are needed to properly concatenate the entire command string.

See Also

Bulk-Copy Functions dbsetopt dbfcmd dbsqlexec dbfreebuf dbsqlsend DB-Library Options

dbcmdrow

Determines whether the current command can return rows.

Syntax

RETCODE dbcmdrow (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED to indicate that the statement can return rows.

FAIL to indicate that the statement can't return rows.

Remarks

dbcmdrow determines whether the current command is a Transact-SQL SELECT statement or an EXECUTE statement on a stored procedure containing a SELECT statement. Call **dbcmdrow** after **dbresults** returns SUCCEED.

Even if **dbcmdrow** returns SUCCEED, the statement will not return any rows if none have qualified. To determine whether any rows are actually being returned, use **dbrows**.

Even if **dbcmdrow** returns FAIL, you must still process the results by calling **dbnextrow** until it returns NO_MORE_ROWS.

See Also

<u>dbnextrow</u>

<u>dbrows</u>

<u>dbresults</u>

<u>dbrowtype</u>

dbcolinfo

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

Syntax

RETCODE dbcolinfo (PDBHANDLE pdbhandle, INT type, DBINT column, DBINT computeid, LPDBCOL lpdbcol);

Arguments

pdbhandle

Is a DBPROCESS pointer or a cursor handle. If *type* is CI_REGULAR or CI_ALTERNATE, this is a DBPROCESS pointer. If *type* is CI_CURSOR, this is a DBCURSOR handle.

type

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

type	Description
CI_REGULAR	Returns information about a regular column in the
	current result set.
CI_ALTERNATE	Returns information about a compute column in the
	current result set.
CI_CURSOR	Returns information about a column in the open
	cursor.

column

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

computeid

If *type* is CI_ALTERNATE, this is the ID that identifies the compute (the result of a COMPUTE clause) value. The *computeid* is returned by **dbnextrow** or **dbgetrow**. If *type* is not CI_ALTERNATE, specify 0.

lpdbcol

Is a pointer to a DBCOL structure that DB-Library will fill with detailed information about the specified column.

The DBCOL structure is defined as follows:

typedef struct

{

DBINT SizeOfStruct; CHAR Name[MAXCOLNAMELEN+1] CHAR ActualName[MAXCOLNAMELEN+1] CHAR TableName[MAXTABLENAME+1] SHORT Type; DBINT UserType; DBINT UserType; DBINT MaxLength; BYTE Precision; BYTE Precision; BYTE Scale; BOOL VarLength; BYTE Null; BYTE CaseSensitive; BYTE Updatable;

- BOOL Identity;
- } DBCOL, PTR LPDBCOL;

The DBCOL fields (and the equivalent DB-Library functions, if any) are described here.

Field	Description
SizeOfStruct	Before calling dbcolinfo , set this field equal to the

	value returned by the C sizeof function for the DBCOL
	structure.
Name	Is the name of the returned column (dbcolname).
ActualName	Is the actual name of the column in TableName if <i>type</i>
	is CI_CURSOR; otherwise, this is the same as Name .
TableName	Is the table that contains the column if <i>type</i> is
	CI_CURSOR; otherwise, NULL.
Туре	Is the data type of the column (dbcoltype , dbalttype).
UserType	Is the user-defined data type of the column
	(dbcolutype, dbaltutype).
MaxLength	Is the maximum length, in bytes, of the column
	(dbcollen, dbaltlen).
Precision	The precision if Type is SQLDECIMAL or
	SQLNUMERIC; otherwise, -1.
Scale	The scale if Type is SQLDECIMAL or
	SQLNUMERIC; otherwise, -1.
VarLength	Is one of the following (dbvarylen):
	TRUE if the column is variable length.
	FALSE if the column is fixed length.
Null	Is one of the following:
	TRUE if the column allows nulls.
	FALSE if the column does not allow nulls.
	DBUNKNOWN if nullability is unknown.
CaseSensitive	Is one of the following:
	TRUE if the column is case-sensitive.
	FALSE if the column is case-insensitive.
	DBUNKNOWN if case sensitivity is unknown.
Updatable	Is one of the following:
	TRUE if the column can be changed.
	FALSE if the column is read-only and cannot be
	changed.
	DBUNKNOWN if updatability is unknown.

Identity	Is one of the following:
	TRUE if the column is an identity column. FALSE if the column is not an identity column.

Returns

SUCCEED or FAIL.

Remarks

Before calling **dbcolinfo**, set the **SizeOfStruct** field equal to the value returned by the C **sizeof** function for the DBCOL structure. The **dbcolinfo** function fills the supplied DBCOL structure with information about the specified column.

Call this function after **dbresults** returns SUCCEED, or after **dbcursoropen** returns a cursor handle.

See Also

<u>dbaltlen</u>

<u>dbcolname</u>

<u>dbalttype</u>

<u>dbcoltype</u>

<u>dbaltutype</u>

<u>dbcolutype</u>

<u>dbcollen</u>

<u>dbvarylen</u>
dbcollen

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

Syntax

DBINT dbcollen (PDBPROCESS dbproc, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the column. The first column is number 1. For further information, see **dbadata**.

Returns

The maximum length of the data for the particular column. If the column number is not in range, -1 is returned.

Remarks

In the case of variable-length data, the maximum length is not necessarily the actual length of the data, but rather the maximum length that the data can be. For the actual data length, use **dbdatlen**. Call **dbcollen** after **dbresults** returns SUCCEED.

Examples

The following example shows how to use **dbcollen**:

```
DBPROCESS *dbproc;
int colnum;
DBINT column_length;
```

```
// Put the command into the command buffer.
dbcmd(dbproc, "SELECT name, id, type FROM sysobjects");
```

// Send the command to SQL Server and begin execution.
dbsqlexec(dbproc);

```
// Process the command results.
dbresults(dbproc);
```

```
// Examine the column lengths.
for (colnum = 1; colnum < 4; colnum++)
{
    column_length = dbcollen(dbproc, colnum);
    printf("column %d, length is %ld.\n", colnum, column_length);
}</pre>
```

See Also

<u>dbcolname</u>

<u>dbdatlen</u>

<u>dbcoltype</u>

<u>dbnumcols</u>

<u>dbdata</u>

dbcolname

Returns a pointer to the null-terminated name of a particular result column.

Syntax

LPCSTR dbcolname (PDBPROCESS dbproc, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the column. The first column is number 1. For further information, see **dbadata**.

Returns

A **char** pointer to the null-terminated name of the particular column. If the column number is not in range or if the column is the result of an expression with no name assigned, **dbcolname** returns NULL. If the column number is out of range, **dbcolname** also calls the error handler with an error of 10011 (SQLECNOR).

Examples

The following example shows how to use **dbcolname**:

DBPROCESS *dbproc;

// Put the command into the command buffer.
dbcmd(dbproc, "SELECT name, id, type FROM sysobjects");

// Send the command to SQL Server and begin execution.
dbsqlexec(dbproc);

// Process the command results.
dbresults(dbproc);

// Examine the column names.
printf("first column name is %s\n", dbcolname(dbproc, 1));
printf("second column name is %s\n", dbcolname(dbproc, 2));
printf("third column name is %s\n", dbcolname(dbproc, 3));

See Also

dbcollen dbdatlen dbcoltype dbnumcols dbdata

dbcoltype

Returns the data type for a regular result column.

Syntax

INT dbcoltype (PDBPROCESS dbproc, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the column. The first column is number 1. For further information, see **dbadata**.

Returns

A value for the data type for the particular column. If the column is not in range, -1 is returned.

Column data type	Returned constant
Char	SQLCHAR
Varchar	SQLCHAR
Binary	SQLBINARY
Varbinary	SQLBINARY
Tinyint	SQLINT1
Smallint	SQLINT2
Int	SQLINT4

Real	SQLFLT4
Float	SQLFLT8
Smallmoney	SQLMONEY4
money	SQLMONEY
decimal	SQLDECIMAL
numeric	SQLNUMERIC
smalldatetime	SQLDATETIM4
datetime	SQLDATETIME
image	SQLIMAGE
text	SQLTEXT

Remarks

The **dbcoltype** function returns an integer value for the type. Use **dbprtype** to convert the type value into a readable string. For more information about a list of SQL Server types, see <u>DB-Library for C Data Types</u>. Call **dbcoltype** after **dbresults** returns SUCCEED.

This function cannot determine whether a column can take null values.

Examples

This example shows how to use **dbcoltype** and **dbprtype**:

DBPROCESS *dbproc; int colnum; int coltype;

// Put the command into the command buffer.
dbcmd(dbproc, "SELECT name, id, type FROM sysobjects");

// Send the command to SQL Server and begin execution.
dbsqlexec(dbproc);

```
// Process the command results.
dbresults(dbproc);
// Examine the column types.
for (colnum = 1; colnum < 4; colnum++)
{
    coltype = dbcoltype(dbproc, colnum);
    printf("column %d, type is %s.\n", colnum, dbprtype(coltype));
}</pre>
```

See Also

<u>dbcollen</u>

DB-Library for C Data Types

<u>dbcolname</u>

<u>dbnumcols</u>

<u>dbdata</u>

<u>dbprtype</u>

<u>dbdatlen</u>

dbcolutype

Returns the user-defined data type for a regular result column.

Syntax

DBINT dbcolutype (PDBPROCESS dbproc, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the column. The first column is number 1. For further information, see **dbadata**.

Returns

The *column* parameter's user-defined data type on success and -1 on error.

Remarks

This function is defined as data type DBINT to accommodate the size of userdefined data types. Both DBINT and user-defined data types are 32 bits long. Call **dbcolutype** after **dbresults** returns SUCCEED.

Example

This example shows how to use **dbcolutype**:

DBPROCESS *dbproc;

int colnum; int numcols;

```
// Put the command into the command buffer.
dbcmd (dbproc, "SELECT * FROM mytable");
```

// Send the command to the SQL Server and begin execution.
dbsqlexec(dbproc);

```
// Process the command results.
dbresults(dbproc);
// Examine the user-defined column types.
numcols = dbnumcols(dbproc);
for (colnum = 1; colnum < numcols; colnum++)
{
    printf
    ("column %d, user-defined type is %ld.\n",
    colnum, dbcolutype(dbproc, colnum));
}</pre>
```

See Also

<u>dbaltutype</u>

<u>dbcoltype</u>

dbconvert

Converts data from one data type to another.

Syntax

INT dbconvert (PDBPROCESS dbproc, INT srctype, LPCBYTE src, DBINT srclen, INT desttype, LPBYTE dest, DBINT destlen);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

In **dbconvert**, the DBPROCESS is used only to supply any custom null values that the program specified through **dbsetnull**. If *dbproc* is NULL, **dbconvert** uses the default values for null value data conversions.

srctype

Is the data type of the source data to be converted. This parameter can be any of the SQL Server data type tokens. You can use **dbcoltype** to get the SQL Server data type token for a particular column.

src

Is a pointer to the source data to be converted. If this pointer is NULL, **dbconvert** puts an appropriate null value in the destination variable. You can use **dbdata** to get a pointer to data from a SQL Server column.

When *srctype* is SQLDECIMAL or SQLNUMERIC, the *src* parameter must be a pointer to a valid DBNUMERIC or DBDECIMAL C variable, respectively.

srclen

Is the length, in bytes, of the source data to be converted. This length is used for the following *srctype* data type tokens:

- SQLCHAR, SQLVARCHAR, SQLTEXT
- SQLBINARY, SQLVARBINARY, SQLIMAGE
- SQLINTN, SQLFLTN, SQLMONEYN, SQLDATETIMN

This length is ignored for all fixed-length, non-NULL data types. If *srclen* is 0, the source data is assumed to be null, and **dbconvert** places an appropriate null value in the destination variable. You can use **dbdatlen** to get the length of SQL Server data.

When *src* points to a DBCHAR string, a *srclen* of -1 indicates that the string is null-terminated.

desttype

Is the destination data type into which the source data is to be converted. This parameter can be any of the SQL Server data type tokens.

dest

Is a pointer to the destination variable that will receive the converted data. If this pointer is NULL, **dbconvert** calls the user-supplied error handler (if any) and returns -1.

When *desttype* is SQLDECIMAL or SQLNUMERIC, the *dest* parameter must be a pointer to a DBNUMERIC or DBDECIMAL C variable, respectively, with the *precision* and *scale* fields of the structure already set to the desired values. You can use DEFAULTPRECISION to specify a default precision and DEFAULTSCALE to specify a default scale.

destlen

Is the length, in bytes, of the destination variable. This length is used for the following *desttype* data type tokens:

- SQLCHAR, SQLVARCHAR, SQLTEXT
- SQLBINARY, SQLVARBINARY, SQLIMAGE
- SQLINTN, SQLFLTN, SQLMONEYN, SQLDATETIMN

The *destlen* is ignored for all fixed-length, non-NULL data types.

When *dest* points to a DBCHAR string or a DBBINARY array, the value of *destlen* must be the total length of the destination buffer space, or -1 to indicate that there is sufficient space available. Note that when *dest* points to a DBCHAR string, a *destlen* of -1 causes the character string to be given a terminating null.

Returns

The length of the converted data, in bytes, if the data type conversion succeeds. If the conversion fails, -1 is returned. If **dbconvert** fails, it first calls a user-supplied error handler (if any). This routine may fail for one of these reasons:

- The requested conversion was unavailable.
- The conversion resulted in truncation, overflow, or loss of precision in the destination variable.
- A syntax error occurred while converting a character string to some numeric type.

Remarks

This function allows the application to convert data from one representation to another. To determine whether a particular conversion is permitted, the program can call **dbwillconvert** before attempting a conversion.

The **dbconvert** function can convert data stored in any of the SQL Server data types (although not all conversions are valid). The following table shows the program variable type (*src* or *dest* parameters) you must provide for each non-NULL SQL Server data type (*srctype* or *desttype* parameters).

Data type (srctype, desttype)	Program variable type (<i>src</i> , <i>dest</i>)
SQLCHAR	DBCHAR
SQLVARCHAR	DBCHAR
SQLTEXT	DBCHAR
SQLBINARY	DBBINARY
SQLVARBINARY	DBBINARY
SQLIMAGE	DBBINARY
SQLINT1	DBTINYINT
SQLINT2	DBSMALLINT
SQLINT4	DBINT
SQLFLT4	DBFLT4
SQLFLT8	DBFLT8
SQLBIT	DBBIT
SQLMONEY4	DBMONEY4
SQLMONEY	DBMONEY
SQLDATETIM4	DBDATETIM4
SQLDATETIME	DBDATETIME
SQLDECIMAL	DBDECIMAL
SQLNUMERIC	DBNUMERIC

The following table shows the program variable type (*src* or *dest* parameters) you must provide for each fixed-length SQL Server data type (*srctype* or *desttype* parameters) that allows NULL values and valid byte length (*srclen* or *destlen* parameters).

srctype, desttype	srclen, destlen	src, dest
SQLINTN	1	DBTINYINT
SQLINTN	2	DBSMALLINT
SQLINTN	4	DBINT

SQLFLTN	4	DBFLT4
SQLFLTN	8	DBFLT8
SQLMONEYN	4	DBMONEY4
SQLMONEYN	8	DBMONEY
SQLDATIMETIMN	4	DBDATETIM4
SQLDATIMETIMN	8	DBDATETIME

Conversion to and from all SQL Server data types is supported, except for conversion between SQLDATETIME or SQLDATETIM4 and the following data types:

- SQLINT1, SQLINT2, SQLINT4
- SQLFLT4, SQLFLT8
- SQLBIT
- SQLMONEY, SQLMONEY4
- SQLDECIMAL, SQLNUMERIC

A conversion to or from the data types SQLBINARY and SQLIMAGE is a straight bit-copy unless the conversion involves SQLCHAR or SQLTEXT. When converting SQLCHAR or SQLTEXT data to SQLBINARY or SQLIMAGE, **dbconvert** interprets the SQLCHAR or SQLTEXT string as hexadecimal digits, whether or not the string contains a leading 0x. When converting SQLBINARY or SQLIMAGE data to SQLCHAR or SQLTEXT, **dbconvert** creates a string of hexadecimal digits without a leading 0x.

Converting to the approximate numeric data types SQLFLT4 or SQLFLT8 can result in some loss of precision. Converting from the approximate numeric data types SQLFLT4 or SQLFLT8 to SQLCHAR or SQLTEXT can also result in some loss of precision.

Converting to SQLFLT*x*, SQLINT*x*, SQLMONEY, SQLMONEY4,

SQLDECIMAL, or SQLNUMERIC can result in overflow if the number is larger than the destination's maximum value, or in underflow if the number is smaller than the destination's minimum value. If overflow occurs when converting to SQLCHAR or SQLTEXT, the first character of the resulting value contains an asterisk (*) to indicate the error.

When binding data to variables rather than accessing the data directly, use **dbbind** to convert instead of **dbconvert**.

For the Microsoft Windows® operating system, DB-Library retrieves information about date, time, numeric, and currency formatting from the Sqlcommn.loc file. The location of Sqlcommn.loc is pointed to by the **SQLLocalizationFile** key in the Windows initialization file (Win.ini) under the [**SQLSERVER**] application heading. For example:

[SQLSERVER] SQLLocalizationFile=C:\SQL60\BIN\SQLCOMMN.LOC

For the Microsoft Windows NT® 4.0 operating system, you set the date, time, numeric, and currency formatting using the International application in Control Panel. Use the SQL Server Client Network Utility **Use International Settings** option to activate this for DB-Library.

Examples

This example converts SQL Server data obtained with **dbdata**:

DBCHAR title[81]; DBCHAR price[9];

// Read the query into the command buffer.
dbcmd(dbproc, "SELECT title, price, royalty FROM pubs..titles");

// Send the query to SQL Server.
dbsqlexec(dbproc);

// Get ready to process the results of the query.
dbresults(dbproc);

```
// Process each row.
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
    // The first dbconvert() adds a null terminator to the string.
    dbconvert(dbproc, SQLCHAR, (dbdata(dbproc,1)), (dbdatlen(dbproc
    SQLCHAR, title, (DBINT)-1);
    // The second dbconvert() converts money to string.
    dbconvert(dbproc, SQLMONEY, (dbdata(dbproc,2)), (DBINT)-1, SC
    price, (DBINT)-1);
```

```
if (dbdatlen(dbproc,3) != 0)
printf ("%s\n $%s %ld\n", title, price,
 *((DBINT *)dbdata(dbproc,3)));
```

```
}
```

See Also

<u>dbaltbind</u>

<u>dberrhandle</u>

<u>dbbind</u>

DB-Library for C Data Types

<u>dbcoltype</u>

<u>dbsetnull</u>

<u>dbdata</u>

<u>dbwillconvert</u>

<u>dbdatlen</u>

Error Messages

dbcount

Returns the number of rows affected by a Transact-SQL statement.

Syntax

DBINT dbcount (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

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

Remarks

After the results of a statement have been processed, you can call **dbcount** to find out how many rows were affected by the statement. For example, if a SELECT statement was sent to SQL Server and you have read all the rows by calling **dbnextrow** until it returned NO_MORE_ROWS, you can call DBCOUNT to find out how many rows were retrieved.

If the current statement doesn't immediately return rows (for example, a DELETE statement), you can call **dbcount** after **dbresults**.

If the current statement executes a stored procedure, for example an EXEC or a call to a remote stored procedure, **dbcount** 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 simply by calling another stored procedure with a SELECT statement.

See Also

<u>dbnextrow</u>

<u>dbresults</u>

DB-Library Options

dbcurcmd

Returns the number of the current command within a command batch.

Syntax

INT dbcurcmd (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The number of the command whose results are currently being processed. The first command in a batch is number 1.

Remarks

The command number is incremented every time **dbresults** returns SUCCEED or FAIL. Unsuccessful commands are counted. The command number is reset by each call to **dbsqlexec** or **dbsqlsend**.

See Also

<u>dbcmdrow</u>

<u>dbrows</u>

<u>dbmorecmds</u>

<u>dbsqlexec</u>

<u>dbresults</u>

<u>dbsqlsend</u>

dbcurrow

Returns the number of the row currently being read.

Syntax

DBINT dbcurrow (PDBPROCESS *dproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The number of the current row.

Remarks

dbcurrow returns the number of the row most recently read by **dbgetrow** or **dbnextrow**. When you do not turn on row buffering, **dbfirstrow**, **dbcurrow**, and **dblastrow** always return the same value (the number of the current row).

Use **dbgetrow** to read rows from the row buffer.

The first row returned from SQL Server is number 1. The row number is changed every time **dbnextrow** or **dbgetrow** returns SUCCEED. The row number is reset to 0 by each new call to **dbresults**.

See Also

<u>dbclrbuf</u>

DB-Library Options

<u>dbfirstrow</u>

<u>dbnextrow</u>

<u>dbgetrow</u>

<u>dbresults</u>

<u>dblastrow</u>

<u>dbsetopt</u>

dbdata

Returns a pointer to the data for a result column.

Syntax

LPCBYTE dbdata (PDBPROCESS dproc, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the column. The first column is number 1. For further information, see <u>dbadata</u>.

Returns

A BYTE pointer to the data for the column. A NULL BYTE pointer is returned if there is no such column or if the data has a null value. To make sure that the data is really a null value, check for a return of 0 from **dbdatlen**.

Remarks

The data is not null-terminated. To get the length of the data, use **dbdatlen**.

The **dbbind** function automatically binds data to your program variables. It is often easier to use than **dbdata**, but it makes a copy of the data. Call **dbdata** only after **dbnextrow** or **dbgetrow** has returned REG_ROW.

Example

The following example shows how to use **dbdata**:

```
DBPROCESS *dbproc;
DBINT row_number = 0;
DBINT object_id;
```

// Put the command into the command buffer.
dbcmd(dbproc, "SELECT id FROM sysobjects");

// Send the command to SQL Server and begin execution.
dbsqlexec(dbproc);

```
// Process the command results.
dbresults(dbproc);
// Examine the data in each row.
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
    row_number++;
    object_id = *((DBINT *)dbdata(dbproc, 1));
    printf("row %ld, object id is %ld.\n", row_number, object_id);
}
```

See Also

<u>dbbind</u>

<u>dbcoltype</u>

<u>dbcollen</u>

<u>dbdatlen</u>

<u>dbcolname</u>

<u>dbnumcols</u>

dbdataready

Determines whether database command processing sent data back to the client is ready to be processed.

Syntax

BOOL dbdataready (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

TRUE if data is available to be read; FALSE otherwise.

Remarks

This function allows an application to continue processing while SQL Server is actually performing the database operation.

The **dbdataready** function is ordinarily used after a call to **dbsqlsend** and before a call to **dbsqlok**. After **dbsqlsend**, SQL Server begins executing the commands in the command buffer. When **dbsqlok** is called, DB-Library for C waits for SQL Server to finish processing before returning control to the application.

The **dbdataready** function provides a way to determine when data is available for processing. It should be called repeatedly until it returns a nonzero value. At that point, the application can call **dbsqlok**, **dbresults**, and **dbnextrow**.

IMPORTANT It is possible for **dbdataready** to return FALSE forever if another process has a conflicting lock or if the connection is broken. The calling program

should contain a time-out mechanism.

See Also

Building Applications

<u>dbsqlok</u>

<u>dbresults</u>

<u>dbsqlsend</u>

<u>dbsettime</u>
dbdatecrack

Converts a computer-readable DBDATETIME value into user-accessible format.

Syntax

RETCODE dbdatecrack (PDBPROCESS dbproc, LPDBDATEREC dateinfo, LPCDBDATETIME datetime);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

dateinfo

Is a pointer to the DBDATEREC structure to contain the parts of *datetime*. DBDATEREC is defined as follows:

typedef struct dbdaterec

{

int year;	// 1753 - 9999
int quarter;	// 1 - 4
int month;	// 1 - 12
int dayofyear;	// 1 - 366
int day;	// 1 - 31
int week;	<pre>// 1 - 54 (for leap years)</pre>
int weekday;	// 1 - 7 (Mon Sun.)
int hour;	// 0 - 23
int minute;	// 0 - 59
int second;	// 0 - 59

int millisecond; // 0 - 999
} DBDATEREC;

datetime

Is a pointer to the DBDATETIME value of interest.

Returns

SUCCEED or FAIL.

Remarks

dbdatecrack converts a DBDATETIME value into its integer components and puts them into a DBDATEREC structure.

DBDATETIME structures store date and time values in an internal format. 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 **dbdatecrack** function converts the internal value to something easily usable by an application.

Using **dbdatecrack** function accepts a NULL value for the *dbproc* parameter. In SQL Server version 6.0 or earlier, the **dbdatecrack** function can only be called if a DBPROCESS structure is active.

Examples

The following code fragment shows how to use **dbdatecrack**:

```
dbcmd(dbproc, "SELECT name, crdate FROM master..sysdatabases");
dbsqlexec(dbproc);
dbresults(dbproc);
```

```
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
    // Print the database name and its date info
    dbconvert(dbproc,
```

```
dbcoltype(dbproc, 2), dbdata(dbproc, 2),
dbdatlen(dbproc, 2), SQLCHAR, datestring, -1);
printf("%s: %s\n", (char *) (dbdata(dbproc, 1)), datestring);
```

// Break up the creation date into its constituent parts
dbdatecrack(dbproc, &dateinfo, (DBDATETIME *) (dbdata(dbproc,

```
// Print the parts of the creation date
printf("\tYear = %d.\n", dateinfo.year);
printf("\tMonth = %d.\n", dateinfo.month);
printf("\tDay of month = %d.\n", dateinfo.day);
printf("\tDay of year = %d.\n", dateinfo.dayofyear);
printf("\tDay of week = %d.\n", dateinfo.weekday);
printf("\tHour = %d.\n", dateinfo.hour);
printf("\tMinute = %d.\n", dateinfo.minute);
printf("\tSecond = %d.\n", dateinfo.second);
printf("\tMillisecond = %d.\n", dateinfo.millisecond);
```

See Also

}

<u>dbconvert</u>

<u>dbdata</u>

dbdatlen

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

Syntax

DBINT dbdatlen (PDBPROCESS dbproc, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the column. The first column is number 1. For further information, see <u>dbadata</u>.

Returns

The actual length of the data for the particular column. If the data has a null value, 0 is returned. If the column number is not in range, -1 is returned.

Remarks

The **dbcollen** function determines the maximum possible length for the data. The data itself is available by calling **dbdata**. Calling **dbdatlen** after **dbnextrow** or **dbgetrow** returns REG_ROW.

Examples

The following example shows how to use **dbdatlen**:

DBPROCESS *dbproc; DBINT row_number = 0; DBINT data_length;

```
// Put the command into the command buffer.
dbcmd(dbproc, "SELECT name FROM sysobjects");
```

// Send the command to SQL Server and begin execution.
dbsqlexec(dbproc);

```
// Process the command results.
dbresults(dbproc);
```

```
// Examine the data lengths of each row.
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
    row_number++;
    data_length = dbdatlen(dbproc, 1);
    printf("row %ld, data length is %ld.\n", row_number, data_length);
}
```

See Also

<u>dbcollen</u> <u>dbdata</u>

<u>dbcolname</u>

<u>dbnumcols</u>

<u>dbcoltype</u>

dbdead

Determines whether a particular DBPROCESS is inactive.

Syntax

BOOL dbdead (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

TRUE or FALSE.

Remarks

dbdead is particularly useful in user-supplied error handlers.

If a DBPROCESS is inactive, almost every DB-Library function that receives it as a parameter immediately fails, calling the user-supplied error handler. You must close an inactive DBPROCESS connection with **dbclose** and open a new connection with **dbopen**.

dbdead usually returns TRUE when a network connection is broken. Note any other errors or messages that you receive.

See Also

<u>dberrhandle</u>

Error Messages

dbenlisttrans

Enlists the current Microsoft® SQL Server[™] 2000 connection in an existing Microsoft Distributed Transaction Coordinator (MS DTC) transaction.

Syntax

RETCODE dbenlisttrans (PDBPROCESS *dbproc*, **LPVOID** *pTransaction*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation/SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

pTransaction

Is the MS DTC transaction OLE object that specifies the transaction to export to SQL Server. The client calls the MS DTC OLE object **ITransactionDispenser::Begin Transaction** method to create the MS DTC transaction object.

Returns

SUCCEED or FAIL.

Remarks

The client application must call **dbenlisttrans** before performing the first update on a server running SQL Server that is enlisted in an MS DTC transaction.

To update two or more (*n*) SQL Servers using MS DTC

1. Connect to the MS DTC by using the MS DTC OLE

DtcSelectTransactionManager method to obtain a transaction manager object.

- 2. Call **dbopen** *n* times to connect to each SQL Server.
- 3. Call the MS DTC OLE **ITransactionDispenser::BeginTransaction** method to begin the MS DTC transaction and to obtain a transaction object.
- 4. Call **dbenlisttrans** *n* times to send the MS DTC transaction to each SQL Server.
- 5. Call **dbsqlexec** *n* times to perform one or more MS DTC transaction updates on each SQL Server.
- 6. Call the MS DTC OLE **ITransaction::Commit** method to commit the MS DTC transaction. The transaction object is no longer valid after the transaction commits.
- 7. Call the MS DTC OLE **ITransaction::Return** method to release the reference to the transaction object.
- 8. Either
 - Call **dbenlisttrans** *n* times to pass a NULL transaction pointer to each SQL Server, which will disconnect each SQL Server.
 - Or
- To begin a new coordinated transaction, go to Step 3.

Note You can also call **dbenlisttrans** and **dbsqlexec** in turn for each SQL Server instead of calling them as suggested in steps 4 and 5 earlier.

See Also

<u>dbenlistxatrans</u>

dbenlistxatrans

Enlists the current Microsoft® SQL Server[™] 2000 connection in a transaction that is coordinated by an XA-compliant transaction processing (TP) monitor.

Syntax

RETCODE debenlistxatrans (PDBPROCESS dbproc, BOOL enlisttran);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

enlisttran

Is TRUE or FALSE. TRUE specifies to enlist in the current XA transaction. FALSE specifies to enlist in the NULL transaction, which results in the connection no longer being enlisted in any XA transaction.

When *enlisttran* is TRUE, the SQL Server connection is enlisted in the current XA transaction. All SQL operations performed using the SQL Server connection are performed under the protection of the current XA transaction.

When *enlisttran* is FALSE, the SQL Server connection is no longer enlisted in any XA transaction. If you have used a SQL Server connection with an XA transaction, and you want to use the same SQL Server connection with a local SQL Server transaction, you must call **dbenlistxatrans** and pass a value of FALSE for the *enlisttran* parameter.

Returns

SUCCEED OR FAIL.

Remarks

To use XA transactions

- 1. Call **dbopen** once for each SQL Server connection to establish.
- Call the X/Open tx_begin function to begin an XA transaction.
 tx_begin calls the appropriate XA-compliant TP monitor.
- 3. For each SQL Server connection you want to enlist in the XA transaction, call **dbenlistxatrans** with *enlisttran* set to TRUE. Each call associates the current XA transaction with the specified SQL Server connection.
- 4. Call **dbsqlexec** one or more times to perform SQL Server operations.
- 5. Call the X/Open **tx_commit** function to commit the XA transaction. To perform a series of XA transactions, repeat steps 2 through 5.

For information about using XA transactions through ODBC, see the SQL_COPT_SS_ENLIST_IN_XA connection option of the **SQLSetConnectOption** function.

See Also

<u>dbenlisttrans</u>

dberrhandle

Supplies a user function to handle DB-Library errors.

Syntax

DBERRHANDLE_PROC dberrhandle (DBERRHANDLE_PROC handler);

Arguments

handler

Is a pointer to the user function called whenever DB-Library determines that an error has occurred. DB-Library calls this function with six parameters:

dbproc

Is the affected DBPROCESS. If there is no DBPROCESS associated with this error, this parameter is NULL.

severity

Is the severity of the error (data type **int**). Error severities are defined in Sqlfront.h.

dberr

Is the identifying number of the error (data type **int**). Error numbers are defined in Sqlfront.h.

oserr

Is the error number that describes the cause of the error (data type **int**) and is specific to the operating system or network. If there is no relevant operating-system error, the value is DBNOERR.

dberrstr

Is a printable description of *dberr* (data type **char ***).

oserrstr

Is a printable description of *oserr* (data type **char ***).

The error handler must return one of the following three values, directing DB-Library to perform particular actions.

Value	Action
INT_EXIT	Prints an error message and exits the application. DB- Library also returns an error to the operating system. With the Microsoft® Windows® operating system, this value is considered an error and is treated as an INT_CANCEL.
INT_CANCEL	Returns FAIL from the DB-Library function that caused the error. For time-out errors (SQLETIME) only, DB-Library calls dbcancel in an attempt to cancel the current command batch and flush any pending results. If this dbcancel attempt also times out, the connection is broken.
INT_CONTINUE	Continues to wait for one additional time-out period, and then calls the error handler again. This return value is meaningful only for time-out errors (SQLETIME). In any other case, this value is considered an error and is treated as an INT_CANCEL.

If the error handler returns any value besides these three, the program continues.

The following example shows a typical error-handling routine:

#include <sqlfront.h>
#include <sqldb.h>

int err_handler(dbproc, severity, dberr, oserr, dberrstr, oserrstr)
DBPROCESS *dbproc;

int severity; int dberr; int oserr; char *dberrstr; char *oserrstr;

{

printf("DB-Library error:\n\t%s\n", dberrstr);

```
if (severity == EXCOMM && (oserr != DBNOERR || oserrstr))
printf(Net-Lib error %d: %s\n''' oserr, oserrstr);
```

```
if (oserr != DBNOERR)
printf("Operating-system error:\n\t%s\n", oserrstr);
if (dbproc == NULL) || (DBDEAD(dbproc))
```

```
return(INT_EXIT);
else
return(INT_CANCEL);
}
```

IMPORTANT Do not call any DB-Library functions from within the error handler because infinite recursive calls to the error handler can result.

Returns

A pointer to the previously installed error handler. This can be NULL.

Remarks

When a DB-Library error occurs, DB-Library immediately calls this error handler. You must install an error handler to handle DB-Library errors properly.

Because the error handler is a callback function, special consideration is required when compiling these functions for the Windows operating system. For more information, see <u>Building Applications</u>. The user-supplied error handler completely determines the response of DB-Library to any error that occurs. It must tell DB-Library which action to take:

• Cancel the application.

- Return an error code.
- Keep trying (in the case of a time-out error).

If the user does not supply an error handler (or passes a null pointer to **dberrhandle**), DB-Library exhibits its default error-handling behavior: It returns FAIL from the DB-Library function that caused the error and program execution continues.

Another function, **dbmsghandle**, installs a message handler that DB-Library calls in response to Microsoft® SQL Server[™] messages. If an application causes messages to occur from DB-Library and SQL Server simultaneously, DB-Library calls the SQL Server message handler before it calls the DB-Library error handler.

The error-handling function should not call any DB-Library functions. Because calls to DB-Library functions can themselves generate errors, calls from within an error handler could result in infinite recursion. If your error handler must call a DB-Library function, it should set the error handler to a null value, and then restore it when it exits.

The following code fragment shows this technique:

```
int err_handler (dbproc, . . .
{
    // Set the error handler to NULL to prevent infinite recursion.
    dberrhandle(NULL);
    // Call other DB_Library functions as necessary.
    .
    .
    // Reset the error handler to this function.
    dberrhandle(err_handler);
    return(. . .
}
```

See Also

<u>dbmsghandle</u>

Error Messages

dbexit

Closes and frees all DBPROCESS structures created as a result of your application.

Syntax

void dbexit (void);

Remarks

The **dbexit** function calls **dbclose** repeatedly for all open DBPROCESS structures. The **dbclose** function cleans up any activity associated with a DBPROCESS structure and frees the space.

You can use **dbclose** directly to close just a single DBPROCESS structure.

See Also

<u>dbclose</u>

<u>dbopen</u>

<u>dbinit</u>

dbfcmd

Adds text to the DBPROCESS command buffer using C run-time library **sprintf**-type formatting.

Syntax

RETCODE dbfcmd (PDBPROCESS dbproc, LPCSTR cmdstring, ... params);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

cmdstring

Is a format string of the form used by the **sprintf** function.

params

Are optional parameters to **dbfcmd**. The number of parameters required depends on the number indicated in the *cmdstring* parameter. The parameters are passed directly to the **sprintf** function.

Returns

SUCCEED or FAIL.

Remarks

This function adds text to the command buffer in the DBPROCESS structure. The **dbfcmd** function works just like the C-language standard I/O library **sprintf** function. If you don't need any of the formatting capability of **sprintf**, use **dbcmd** instead.

Conversion	Program variable type
%с	char
%s	null-terminated string
%d	int
%f	double
%g	double
%e	double
%%	none
%с	char
%x	unsigned hexadecimal integer string
%u	unsigned decimal

The following table lists the conversions supported by **dbfcmd**.

The data type SQLDATETIME must be converted into a character string and passed, using %s. SQLMONEY can be converted to a character string and passed, using %s, or it can be converted to **float** and passed, using %f.

To include a percent character (%) in the command string, encode it as two percent characters (%%) because **dbfcmd** calls **sprintf**, which treats the % character as a format specification. In addition, don't use variables containing strings with apostrophes or single quotation marks because they conflict with the SQL statement syntax if there is any. If you don't need any of the formatting capability of **sprintf**, you can use **dbcmd**.

The **dbfcmd** function manages the space allocation for the command buffer. It adds to the existing command buffer; it doesn't delete or overwrite the current contents except after the buffer has been sent to SQL Server. You can call **dbfcmd** repeatedly. Note that sequential calls are concatenated: The application must make sure that any necessary blanks appear between the end of one line and the beginning of the next.

After a call to **dbsqlexec** or **dbsqlsend**, the first call to either **dbcmd** or **dbfcmd** automatically clears the command buffer before the new text is entered. If this

situation is undesirable, set the DBNOAUTOFREE option. When DBNOAUTOFREE is set, the command buffer is cleared only by a call to **dbfreebuf**.

Do not pass **dbfcmd** null pointers contained in variables.

An application can intermingle calls to **dbcmd** and **dbfcmd**.

At any time, an application can access the contents of the command buffer through calls to **dbgetchar**, **dbstrlen**, and **dbstrcpy**.

Available memory is the only constraint on the size of the DBPROCESS command buffer created by calls to **dbcmd** and **dbfcmd**.

This function does not support the C data type **long double**.

This function allocates its working buffer dynamically. The size it picks to allocate space is the maximum of a defined constant (1024) and the string length of *cmdstring* * 2. If the arguments are big in comparison to the size of *cmdstring*, DB-Library may not be able to allocate enough space.

Examples

The following example shows how to use **dbfcmd** to build up a multiple-line SQL command. Note the leading spaces.

char *column_name; DBPROCESS *dbproc; int low_id; char *object_type; char *tablename;

```
dbfcmd(dbproc, "SELECT %s FROM %s", column_name, tablename)
dbfcmd(dbproc, "WHERE id > %d", low_id);
dbfcmd(dbproc, "AND type='%s'", object_type);
```

See Also

Bulk-Copy Functions

<u>dbfreebuf</u>

<u>dbcmd</u>

DB-Library Options

dbfirstrow

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

Syntax

DBINT dbfirstrow (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The number of the first row in the row buffer. The first row returned from SQL Server is number 1. Use this return value when making a call to **dbgetrow**.

Remarks

If you are not buffering rows, **dbfirstrow**, **dbcurrow**, and **dblastrow** always have the same value: the current row number in the current batch. If you enable buffering by setting the DBBUFFER option, **dbfirstrow** returns the number of the row that is the first row in the row buffer. For example, if a query returns 100 result rows and you read 20 rows into the buffer using **dbnextrow**, **dbfirstrow** returns 1, regardless of which row is current. As the application processes and clears rows from the buffer using **dbclrbuf** and more data is read from the server using **dbnextrow**, **dbfirstrow** returns the number of the result row stored in the lowest (oldest) buffer location.

dblastrow returns the number of the result row stored in the highest (newest) buffer location.

See Also

Bulk-Copy Functions

<u>dblastrow</u>

<u>dbclrbuf</u>

DB-Library Options

<u>dbcurrow</u>

<u>dbnextrow</u>

<u>dbgetrow</u>

<u>dbsetopt</u>

dbfreebuf

Clears the command buffer.

Syntax

void dbfreebuf (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Remarks

This function frees any space allocated to the command buffer of the DBPROCESS structure. The command buffer is then set to NULL. Commands for SQL Server are added to the command buffer with the **dbcmd** or **dbfcmd** function.

After a call to **dbsqlexec** or **dbsqlsend**, the first call to either **dbcmd** or **dbfcmd** automatically calls **dbfreebuf** to clear the command buffer before the new text is entered. If this situation is undesirable, set the DBNOAUTOFREE option. When DBNOAUTOFREE is set, the command buffer is cleared only by a call to **dbfreebuf**.

The contents of the command buffer can be accessed through the **dbgetchar**, **dbstrlen**, and **dbstrcpy** functions.

See Also

Bulk-Copy Functions
dbsqlexec

<u>dbcmd</u>

<u>dbsqlsend</u>

<u>dbfcmd</u>

<u>dbstrcpy</u>

<u>dbgetchar</u>

<u>dbstrlen</u>

DB-Library Options
dbfreelogin

Frees a login record.

Syntax

void dbfreelogin (PLOGINREC login);

Arguments

login

Is the pointer to a LOGINREC structure returned from the **dblogin** function.

Remarks

This function frees the memory allocated by the **dblogin** function. It can be called immediately after a call to **dbopen**; however, you can use the same login record for multiple calls to **dbopen**. Call **dbfreelogin** when you are completely finished with the login record.

Examples

The following example shows how to use **dbfreelogin**:

```
DBPROCESS *dbproc;
LOGINREC *loginrec;
```

```
loginrec = dblogin();
DBSETLUSER(login, "user");
DBSETLPWD(loginrec, "my_password");
DBSETLAPP(loginrec, "my_program");
dbproc = dbopen(loginrec, "my_server");
dbfreelogin (loginrec);
```

See Also

<u>dblogin</u>

<u>dbopen</u>

dbgetchar

Returns a pointer to a character in the command buffer.

Syntax

LPSTR dbgetchar (PDBPROCESS *dbproc*, INT *n*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

n

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

Returns

A pointer to the *n*th character in the command buffer. If *n* is not in range, NULL is returned.

Remarks

The **dbgetchar** function can be used to find a particular character in the command buffer. It returns a pointer to the *n*th character in the command buffer.

Internally, the command buffer is a linked list of nonnull-terminated text strings. Parts of the command buffer can be located and copied using the **dbgetchar**, **dbstrcpy**, and **dbstrlen** functions.

Because the command buffer is not just one large text string, but rather a linked

list of text strings, you must use **dbgetchar** to index through the buffer. If you just get a pointer using **dbgetchar** and then increment it yourself, it will probably fall off the end of a string.

See Also

<u>dbcmd</u>

<u>dbstrcpy</u>

<u>dbfcmd</u>

<u>dbstrlen</u>

<u>dbfreebuf</u>

dbgetmaxprocs

Determines the current maximum number of simultaneously open DBPROCESS structures.

Syntax

SHORT dbgetmaxprocs (void);

Returns

An integer representing the current limit on the number of simultaneously open DBPROCESS structures.

Remarks

A DB-Library for C program has a maximum number of simultaneously open DBPROCESS structures. By default, this number is 25. The application program can change this limit by calling **dbsetmaxprocs**.

See Also

<u>dbopen</u>

<u>dbsetmaxprocs</u>

dbgetoff

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

Syntax

INT dbgetoff (PDBPROCESS dbproc, DBUSMALLINT offtype, INT startfrom);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL ServerTM 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

offtype

Is the type of offset you want to find. The types (defined in the header file Sqlfront.h) are: OFF_SELECT, OFF_FROM, OFF_ORDER, OFF_COMPUTE, OFF_TABLE, OFF_PROCEDURE, OFF_STATEMENT, OFF_PARAM, and OFF_EXEC

For details, see <u>DB-Library Options</u>.

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 the specified offset. If the offset is not found, -1 is returned.

Remarks

If the DBOFFSET option has been set, **dbgetoff** can check for the location of certain Transact-SQL statements in the command buffer.

Examples

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

```
int select_offset[10];
int last_offset;
int i;
```

```
// Set the offset option.
dbsetopt(dbproc, DBOFFSET, "select");
```

dbsqlexec(dbproc); // Execute the option on the server while(dbresi // Assume the command buffer contains the following SELECTs: dbcmd(dbproc, "SELECT x = 100 SELECT y = 5");

```
// Send the query to SQL Server.
dbsqlexec(dbproc);
```

```
// Get all the offsets to the SELECT keyword.
for (i = 0, last_offset = 0; last_offset != -1; i++)
if ((last_offset = dbgetoff(dbproc, OFF_SELECT, last_offset))!= -1)
    select_offset[i] = last_offset++;
```

```
dbresults(dbproc);
```

In this example, select_offset[0] = 0 and select_offset[1] = 15.

The function **dbgetoff** does not recognize SELECT statements in a subquery. If the command buffer contains the following, the second SELECT statement goes

unrecognized:

SELECT pub_name FROM publishers WHERE pub_id NOT IN (SELECT pub_id FROM titles WHERE type = "business")

See Also

Bulk-Copy Functions

<u>dbsetopt</u>

<u>dbcmd</u>

<u>dbstrcpy</u>

<u>dbgetchar</u>

<u>dbstrlen</u>

DB-Library Options

dbgetpacket

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

Syntax

UINT dbgetpacket (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The TDS packet size currently in use.

Remarks

To determine the TDS packet size in use, an application should call **dbgetpacket** after **dbopen**.

You can change the TDS packet size by using **DBSETLPACKET**, which sets the packet size field in the LOGINREC structure.

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

See Also

<u>dbsetlpacket</u>

dbgetrow

Reads the specified row in the row buffer.

Syntax

STATUS dbgetrow (PDBPROCESS dbproc, DBINT row);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

row

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

Returns

One of four different types of values:

- If the current row is a regular row, REG_ROW is returned.
- If the current row is a compute row, the *computeid* of the row is returned (for information about the *computeid*, see <u>dbaltbind</u>).
- If the row is not in the row buffer, NO_MORE_ROWS is returned.
- If the function was unsuccessful, FAIL is returned.

Remarks

The **dbgetrow** function sets the current row to a specific row and reads it. This function only works if the DBBUFFER option is on. Any specified binding of row data to program variables takes effect.

When buffering is not turned on, generally each row is processed in turn by repeatedly calling **dbnextrow** until it returns NO_MORE_ROWS. When buffering is turned on, **dbgetrow** allows the user to jump to any row that has already been read using **dbnextrow** and is still in the row buffer. Calls to **dbnextrow** after a **dbgetrow** call return rows in order, following the row read by **dbgetrow**.

See Also

Bulk-Copy Functions DB-Library Options dbbind dbnextrow dbclrbuf

dbgettime

Returns the number of seconds that DB-Library waits for Microsoft® SQL Server[™] to respond to a Transact-SQL statement.

Syntax

INT dbgettime (void);

Returns

The time-out value is the number of seconds that DB-Library waits for a SQL Server response before timing out. A time-out value of 0, the default, represents an infinite time-out period.

Remarks

dbgettime returns the length of time, in seconds, that DB-Library waits for a SQL Server response during calls to **dbsqlexec**, **dbsqlok**, **dbresults**, and **dbnextrow**.

The program can call **dbsettime** to change the time-out value.

See Also

<u>dbnextrow</u>

<u>dbsqlexec</u>

<u>dbresults</u>

<u>dbsqlok</u>

<u>dbsettime</u>

dbgetuserdata

Returns a pointer to user-allocated data from a DBPROCESS structure.

Syntax

LPVOID dbgetuserdata (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

A generic BYTE pointer to the user's private data space. This pointer must have been previously saved with the **dbsetuserdata** function.

Remarks

The functions **dbgetuserdata** and **dbsetuserdata** allow the application to associate user data with a particular DBPROCESS, without using global variables. One use for these functions is to handle deadlock. (For an example, see the example under **dbsetuserdata**. That example reruns the transaction when the application's message handler detects deadlock.)

You can call **dbgetuserdata** when **dbdead** returns TRUE.

This function is particularly useful when the application has more than one DBPROCESS.

See Also

<u>dbsetuserdata</u>

dbinit

Initializes DB-Library.

Syntax

LPCSTR dbinit (void);

Returns

A null-terminated string containing the version of DB-Library, or NULL if initialization is unsuccessful.

Remarks

The application should call **dbinit** once and only once, before calling any other DB-Library functions.

The **dbinit** function enables **dbconvert** to use the international settings specified in Sqlcommn.loc for Microsoft® Windows® 2000 or in the Microsoft Windows NT® 4.0 Registry.

For the Windows operating system, DB-Library retrieves information about date, time, numeric, and currency formatting from the Sqlcommn.loc file. The location of Sqlcommn.loc is pointed to by the **SQLLocalizationFile** key in the Windows initialization file (Win.ini) under the [**SQLSERVER**] application heading. For example:

[SQLSERVER] SQLLocalizationFile=C:\SQL60\BIN\SQLCOMMN.LOC

For the Windows NT 4.0 operating system, you set the date, time, numeric, and currency formatting using the International application in the Control Panel.

Under Microsoft MS-DOS®, **dbinit** detects the presence of the Net-Library TSR used by DB-Library to communicate over the network. If the TSR is not detected, **dbinit** returns NULL and DB-Library functions cannot be used. 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 **dbinit**; it does this to prevent conflicts between applications that use DB-Library concurrently. For DB-Library to release this information, the application must call **dbwinexit** just before it exits.

See Also

<u>dbconvert</u> <u>dbwinexit (Windows only)</u>

dbisavail

Determines whether a DBPROCESS structure is available for general use.

Syntax

BOOL dbisavail (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

TRUE if DBPROCESS is available for general use; FALSE otherwise.

Remarks

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

See Also

<u>dbsetavail</u>

dbiscount

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

Syntax

BOOL dbiscount (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL ServerTM 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server. Unlike the functions **dbsetopt** and **dbclropt**, *dbproc* cannot be NULL here.

Returns

TRUE if the count returned by **dbcount** is real or FALSE if the count returned by **dbcount** 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, **dbcount** returns 0.

Calling **dbiscount** after **dbcount** determines whether the count is real.

See Also

<u>dbcount</u>

dbisopt

Checks the status of a Microsoft[®] SQL Server[™] 2000 or DB-Library option.

Syntax

BOOL dbisopt (PDBPROCESS dbproc, INT option, LPCSTR param);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server. Unlike the functions **dbsetopt** and **dbclropt**, *dbproc* cannot be NULL here.

option

Is the option to be checked.

param

Is the *option* parameter. Certain options take parameters. If an option does not take a parameter, *param* is ignored. If an option does take a parameter, *param* is ignored for all options except DBOFFSET and DBSTAT.

The DBOFFSET and DBSTAT options can have several settings, each with a different parameter. In these cases, **dbisopt** needs a valid *param* to determine which option parameter to check.

For more information about a list of options that take parameters, see <u>DB-Library Options</u>.

Returns

TRUE or FALSE.

Remarks

Although SQL Server options can be set and cleared directly through Transact-SQL, the application should use **dbsetopt** and **dbclropt** to set and clear options. This provides a uniform interface for setting both SQL Server and DB-Library options. It also allows the application to use the **dbisopt** function to check the status of an option.

For more information about a list of each option and its default status, see <u>DB-Library Options</u>.

See Also

Bulk-Copy Functions DB-Library Options dbclropt dbsetopt

dblastrow

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

Syntax

DBINT dblastrow (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

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

Remarks

If you aren't buffering rows, **dbfirstrow**, **dbcurrow**, and **dblastrow** always have the same value. If you have enabled buffering by setting the DBBUFFER option, **dblastrow** returns the number of the last row in the row buffer, which may not be the last row available from the server. Use this return value with a call to **dbgetrow**. For example, if a query returns 100 result rows and you read 20 rows into the buffer using **dbnextrow**, **dblastrow** returns 20 regardless of which row is current. As the application processes and clears rows from the buffer using **dbclrbuf** and more data is read from the server using **dbnextrow**, **dblastrow** returns the number of the result row stored in the highest (newest) buffer location.

See Also

Bulk-Copy Functions

<u>dbgetrow</u>

<u>dbclrbuf</u>

DB-Library Options

<u>dbcurrow</u>

<u>dbnextrow</u>

<u>dbfirstrow</u>

<u>dbsetopt</u>

dblogin

Allocates a LOGINREC structure for use in **dbopen**.

Syntax

PLOGINREC dblogin (void);

Returns

A pointer to a LOGINREC structure. NULL is returned if the structure cannot be allocated.

Remarks

DB-Library contains various functions that supply components of the LOGINREC.

Function	Component supplied
DBSTLUSER	Login ID.
DBSETLPWD	Password.
DBSETLHOST	Workstation name.
DBSETLAPP	Application name.
DBSETLNATLANG	National language.
DBSETLPACKET	TDS packet size.
DBSETLSECURE	Secure connection request.
DBSETLTIME	Connection-specific time-out.
DBSETLVERSION	DB-Library 6.0 behavior.
BCP_SETL	Enables bulk copy operations.

Generally, it is necessary for the program to supply only the username and password. All components in the LOGINREC are initially set to NULL.

See Also

BCP_SETL

<u>dbsetlpacket</u>

<u>dbfreelogin</u>

<u>dbsetlpwd</u>

<u>dbopen</u>

<u>dbsetlsecure</u>

<u>dbsetlapp</u>

<u>dbsetltime</u>

<u>dbsetlhost</u>

<u>dbsetluser</u>

dbsetlnatlang

<u>dbsetlversion</u>
dbmorecmds

Indicates whether there are more commands to be processed.

Syntax

RETCODE dbmorecmds (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED to indicate more results in the command batch; otherwise, FAIL.

Remarks

It should be called after **dbnextrow** returns NO_MORE_ROWS. If you know that the current command does not return any rows, you can call **dbmorecmds** after **dbresults** returns SUCCEED.

You can get the same information by calling **dbresults** until it returns NO_MORE_RESULTS.

See Also

<u>dbcmdrow</u>

<u>dbrows</u>

<u>dbnextrow</u>

<u>dbrowtype</u>

<u>dbresults</u>

dbmsghandle

Installs a user function to handle Microsoft[®] SQL Server[™] 2000 messages.

Syntax

DBMSGHANDLE_PROC dbmsghandle (DBMSGHANDLE_PROC handler);

Arguments

handler

Is a pointer to the user function that is called whenever DB-Library receives an error or informational message from SQL Server. DB-Library calls this function with these parameters:

dbproc

Is the affected DBPROCESS.

msgno

Is the online message number (data type DBINT). These numbers are documented in Troubleshooting. They are stored online in the **sysmessages** table, from which they can be selected.

msgstate

Is the message error state number (data type **int**). These numbers provide information about the context of the error.

severity

Is the message information class or error severity (data type **int**). These numbers are documented in Troubleshooting.

msgtext

Is the message null-terminated text (data type **char** *).

srvname

Is the null-terminated name of the server that generated the message (data type **char** *). A server name is stored in the **srvname** column of its **sysservers** system table. *srvname* is optional.

procname

Is the null-terminated name of the stored procedure that generated the message (data type **char** *). If the message was not generated by a stored procedure, *procname* is set to a length of 0. *procname* is optional.

line

Is the number of the command batch or stored procedure line that generated the message (data type *DBUSMALLINT*). Line numbers start at 1. *line* is optional.

The line number pertains to the nesting level at which the message was generated. For instance, if a command batch executes stored procedure A, which then calls stored procedure B, and a message is generated at line 3 of B, then the value of *line* becomes 3.

If no line number is associated with the message, *line* becomes 0. Circumstances that can generate messages without line numbers include a login error or a call to a remote procedure (through **dbrpcsend**) to a stored procedure that doesn't exist.

The message handler must return a value of 0 to DB-Library.

Returns

A pointer to the previously installed message handler. This can be NULL.

Remarks

This function installs a message-handling function that you supply. When DB-Library receives a SQL Server error or informational message, it immediately calls this message handler. You must install a message handler to handle SQL Server messages.

Because the message handler is a callback function, special considerations are required when compiling these functions under the Microsoft Windows® operating system. For more information, see <u>Building Applications</u>.

Note The *srvname*, *procname*, and *line* parameters are optional, and compatibility with earlier versions of SQL Server is maintained through the **cdecl** parameter declaration convention.

Examples

The following example shows a typical message-handling function:

```
#include <sqlfront.h>
#include <sqldb.h>
```

```
int msg_handler (dbproc, msgno, msgstate, severity, msgtext, srvname,
                     *dbproc;
DBPROCESS
DBINT
                 msgno
             msgstate
int
int
             severity
char
                *msgtext
                *srvname
char
char
                *procname
DBUSMALLINT
                       line
{
 printf ("SQL Server message %ld, state %d, "
   "severity %d: \n\t%s\n", msgno, msgstate, severity, msgtext);
 if (strlen (srvname) != 0)
   printf ("Server '%s', ", srvname);
 if (strlen (procname) != 0)
   printf ("Procedure '%s', ", procname);
 if (line !=0)
   printf ("Line %d", line);
return (0);
}
```

See Also

<u>dberrhandle</u>

dbname

Returns the name of the current database.

Syntax

LPCSTR dbname (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

A pointer to the null-terminated name of the current database.

Remarks

If you need to keep track of when the database changes, use **dbchange**.

See Also

<u>dbchange</u>

dbnextrow

Reads in the next row.

Syntax

STATUS dbnextrow (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

One of five different types of values:

• If a regular row is read, REG_ROW is returned.

Regular rows contain data from columns designated by a SELECT statement.

- If a compute row is read, the *computeid* of the row is returned (for information about *computeid*, see <u>dbaltbind</u>).
- If there are no more rows to be read, if the statement didn't return any rows, or if the server was unable to return more rows (for example, when a deadlock occurs), NO_MORE_ROWS is returned.
- If buffering is turned on and reading the next row would cause the buffer to be exceeded, BUF_FULL is returned.

In this case, no row has been read. To read more rows, first clear at least one row from the top of the row buffer. To clear the row buffer, call

dbclrbuf.

• If the function was unsuccessful, FAIL is returned.

Remarks

The **dbnextrow** function causes the next data row to be made available through the *dbproc*. If the DBBUFFER option is turned on and rows have been read out of order by calling **dbgetrow**, the next data row is read from the buffered rows. Any specified binding of row data to program variables takes effect.

The **dbresults** function must be called and must have returned SUCCEED before you make any calls to **dbnextrow**.

Even if **dbrows** or **dbcmdrow** returns FAIL (indicating that no rows were returned), you must process the results by calling **dbnextrow** until it returns NO_MORE_ROWS.

Normally, each row is processed in turn by repeatedly calling **dbnextrow**. If row buffering is enabled and the row buffer has been cleared by the **dbclrbuf** function, the discarded rows are no longer available (even if **dbgetrow** tries to position to a discarded row). When row buffering is disabled, the last row is cleared when **dbnextrow** returns NO_MORE_ROWS.

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, **dbnextrow** returns different values according to the type of row.

If you want data returned from SQL Server to be displayed on the default output device, use **dbprrow** instead of **dbnextrow** (except with the Microsoft Windows® operating system).

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 **dbdataready** before **dbnextrow** and set the DB-Library time-out to regain control periodically.

Examples

The typical sequence of calls is:

```
DBINT xvariable;
DBCHAR yvariable[10];
```

```
// Read the query into the command buffer.
dbcmd(dbproc, "SELECT x = 100, y = 'hello'");
```

```
// Send the query to SQL Server.
dbsqlexec(dbproc);
```

```
// Get ready to process the results of the query.
dbresults(dbproc);
```

```
// Bind column data to program variables.
dbbind(dbproc, 1, INTBIND, (DBINT) 0, (BYTE *)&xvariable);
dbbind(dbproc, 2, STRINGBIND, (DBINT) 0, yvariable);
```

```
// Now process each row.
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
    // C-code to print or process row data
}
```

Note that if you are not using row buffering, you must continue calling **dbnextrow** until it returns NO_MORE_ROWS. This is true even if you are sure that your query only generates one results row. The **while** loop in the preceding example illustrates the correct way to use **dbnextrow**.

See Also

Bulk-Copy Functions

<u>dbgetrow</u>

<u>dbbind</u>

DB-Library Options

<u>dbclrbuf</u>

<u>dbprrow</u>

<u>dbresults</u>

dbnullbind

Associates an indicator variable with a regular result row column.

Syntax

RETCODE dbnullbind (PDBPROCESS dbproc, INT column, LPCDBINT indicator);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the column to be associated with the indicator variable.

indicator

Is a pointer to the indicator variable.

Returns

SUCCEED or FAIL. Returns FAIL if column is invalid.

Remarks

The indicator variable reveals whether a column in a particular regular result row has been converted and copied to a program variable successfully, or whether it is NULL. Call **dbnullbind** after **dbresults** has returned SUCCEED, and after calling **dbbind** for *column*.

The indicator variable is set when regular result rows are processed through

dbnextrow. The possible values include:

- -1 if the column is NULL.
- The full length of column's data, in bytes, if *column* was bound to a program variable through **dbbind**, the binding did not specify any data conversions, and the bound data was truncated because the program variable was too small to hold data for *column*.
- 0 if *column* was bound and copied to a program variable successfully.

Note Only CHARBIND and VARYCHARBIND can detect truncation of character strings.

See Also

<u>dbanullbind</u> <u>dbdatlen</u> <u>dbbind</u>

dbnextrow

<u>dbdata</u>

dbnumalts

Returns the number of columns in a compute row.

Syntax

INT dbnumalts (PDBPROCESS dbproc, INT computeid);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

computeid

Is the ID that identifies 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 **dbnextrow** or **dbgetrow**.

Returns

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

Remarks

Call this function after **dbresults** has returned SUCCEED.

Examples

In this example, **dbnumalts(dbproc, 1)** returns 3:

SELECT dept, year, sales FROM employee ORDER BY dept, year COMPUTE AVG(sales), MIN(sales), MAX(sales) BY dept

See Also

dbadatadbgetrowdbadlendbnextrowdbaltlendbresultsdbalttype

dbnumcols

Determines the number of columns for the current result set.

Syntax

INT dbnumcols (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The number of columns in the current result set. If there are no columns, 0 is returned. Call **dbnumcols** after **dbresults** has returned SUCCEED.

Examples

The following program fragment shows how to use **dbnumcols**:

int column_count;
DBPROCESS *dbproc;

// Put the commands into the command buffer dbcmd(dbproc, "SELECT name, id, type FROM sysobjects"); dbcmd(dbproc, " SELECT name FROM sysobjects");

// Send the commands to SQL Server and start execution
dbsqlexec(dbproc);

```
// Process each command until there are no more
while (dbresults(dbproc) != NO_MORE_RESULTS)
{
    column_count = dbnumcols(dbproc);
    printf("%d columns in this SQL Server result.\n", column_count);
    while (dbnextrow(dbproc) != NO_MORE_ROWS)
        printf("row received.\n");
}
```

See Also

<u>dbcollen</u>

<u>dbcolname</u>

dbnumcompute

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

Syntax

INT dbnumcompute (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

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

Remarks

Call this function after **dbresults** has returned SUCCEED.

Example

In this example, **dbnumcompute** (*dbproc*) 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

<u>dbnumalts</u>

<u>dbresults</u>

dbnumorders

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

Syntax

INT dbnumorders (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The number of ORDER BY columns. If there is no ORDER BY clause, **dbnumorders** returns 0. If there is an error, **dbnumorders** returns -1.

Remarks

After a SELECT statement has been executed and **dbresults** has been called to process it, you can call **dbnumorders** to find out how many columns were specified in the current statement's ORDER BY clause. Call **dbnumorders** after **dbresults** has returned SUCCEED.

See Also

<u>dbordercol</u>

<u>dbresults</u>

dbopen

Allocates and initializes a DBPROCESS connection structure.

Syntax

PDBPROCESS dbopen (PLOGINREC login, LPCSTR servername);

Arguments

login

Is a pointer to a LOGINREC structure. You can get one by calling **dblogin**.

servername

Is the name of the server running Microsoft® SQL ServerTM 2000 to connect to. This parameter can be the name of an actual SQL Server, NULL or a null string to connect to a local SQL Server, or the logical name matching an entry in the Win.ini file or the Microsoft Windows NT® 4.0 Registry.

Returns

A DBPROCESS pointer if everything is correct. Ordinarily, NULL is returned if a DBPROCESS structure could not be created or initialized, or if your login to SQL Server failed. When NULL is returned, the user-supplied error handler is called to indicate the error.

Note If there is an unexpected communications failure during the SQL Server login process and an error handler has not been installed, the function returns NULL.

Errors

The **dbopen** function returns NULL if any of the following are true.

Error code	Description

SQLEMEM	Unable to allocate sufficient memory.
SQLEDBPS	Maximum number of DBPROCESS structures already
	allocated.
SQLECONN	Unable to connect: SQL Server is unavailable or does
	not exist.
SQLEPWD	Login incorrect.

If the specified server cannot be found, a SQLECONN error is returned to your error handler (if one was registered) after the login time-out expires. For more information about setting the login time-out, see **dbsetlogintime**.

Remarks

The DBPROCESS structure is the basic data structure that DB-Library uses to communicate with SQL Server. The application needs a DBPROCESS structure to communicate with SQL Server. It is the first parameter in almost every DB-Library call. Besides allocating the DBPROCESS structure, this function sets up communication with the network, logs in to SQL Server, and initializes any default options.

In the call to **dbopen**, DB-Library uses the server name and connection information in the [**SQLSERVER**] section of Win.ini, or the following subtree of the Windows NT 4.0 Registry:

```
HKEY_LOCAL_MACHINE\
SOFTWARE\
Microsoft\
MSSQLServer\
Client\
ConnectTo
```

Use the SQL Server Client Network Utility to configure server name and connection information.

Note also that the SQL Server ODBC driver uses the same Net-Library mechanism as DB-Library.

There can be only one Net-Library TSR loaded for Microsoft MS-DOS®, so

there is no .ini file configuration. Environment variables for the name of the server are used to specify any network-specific connection information.

You can use **dbopen** to connect to a failover server. When used to connect to a failover server, this function only works with a standby server configuration; it does not apply to a virtual server cluster/failover configration. If an attempt to connect to a primary server fails, **dbopen** can attempt to connect to a failover server. You must call the **DBSETLFALLBACK** function to indicate that a failover server can be used. The call to **DBSETLFALLBACK** must occur before the call to **dbopen**.

Each call to **DBSETLFALLBACK** from a computer running Windows NT 4.0, Microsoft Windows® 95 or Windows 98 updates the failover information in the local registry from the SQL Server registry. Failover information is stored in the Win.ini file on computers running 16-bit Windows. The **dbopen** function retrieves the failover information from the local registry or from the Win.ini file.

When **dbopen** attempts to connect to a failover server, DB-Library generates warning message 10110 (SQLECONNFB):

Unable to connect: DB Server is unavailable or does not exist - will attempt a fallback connection.

Example

The following example shows how to use **dbopen**:

```
DBPROCESS *dbproc;
LOGINREC *loginrec;
```

```
loginrec = dblogin();
DBSETLUSER(loginrec, "user");
DBSETLPWD(loginrec, "my_password");
DBSETLAPP(loginrec, "my_program");
dbproc = dbopen(loginrec, "my_server");
```

See Also

dbclosedbsetlfallbackdbexitdbsetlogintimedblogin

dbordercol

Returns the ID of a column appearing in the most recently executed query's ORDER BY clause.

Syntax

INT dbordercol (PDBPROCESS dbproc, INT order);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

order

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

Returns

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

Remarks

Call **dbordercol** after **dbresults** has returned SUCCEED.

Example

In this example, **dbordercol**(*dbproc*, 1) returns 3 since the first column named in
the ORDER BY clause refers to the third column in the SELECT statement:

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

See Also

<u>dbnumorders</u>

dbprhead

Prints the column headings for rows returned from Microsoft® SQL Server[™] 2000.

Syntax

void dbprhead (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Remarks

This function displays the column headings for a set of query results on the default output device in a format similar to that used by Isql.exe. The format is compatible with the format used by **dbprrow**. The output width is set at 80 columns.

You can call **dbprhead** after **dbresults** has succeeded.

The **dbprhead** function is useful for debugging.

The **dbprhead** function is not supported with the Microsoft Windows® operating system.

See Also dbbind dbprrow dbnextrow <u>dbresults</u>

dbprocerrhandle

Supplies a user function to handle DB-Library errors for a specific DBPROCESS connection.

Syntax

DBERRHANDLE_PROC dbprocerrhandle (PDBHANDLE *pdbhandle,* **DBERRHANDLE_PROC** *error_handler* **);**

Arguments

pdbhandle

Is a DBPROCESS structure or a LOGINREC structure. If this is a DBPROCESS connection, the *error_handler* will be used for that connection. If this is a LOGINREC structure, the *error_handler* will automatically be used for all future DBPROCESS connections opened (with **dbopen**) using the LOGINREC.

error_handler

A pointer to the user function that is called whenever DB-Library determines that an error has occurred for the existing or future connection specified in *pdbhandle*. Define the connection error-handler function in exactly the same way as the application error-handler function specified in **dberrhandle**.

Returns

If *pdbhandle* is a DBPROCESS structure, a pointer to the previously installed connection error handler is returned. This can be NULL.

If *pdbhandle* is a LOGINREC structure, a pointer to the newly installed connection error handler is returned, or NULL if this function fails.

Remarks

This function is similar to the **dberrhandle** function. While **dberrhandle** installs an error handler global to the entire DB-Library application,

dbprocerrhandle installs an error handler for a specific DBPROCESS connection.

When a DB-Library error occurs using a connection that has a connection error handler (installed using **dbprocerrhandle**), only the connection error handler is called. The application error handler is not called. Because a connection error handler is associated with a connection and is not global to the entire DB-Library application, the connection error-handler code does not need to protect against reentrancy by DB-Library.

Note that this function is not supported for Microsoft MS-DOS®.

See Also

<u>dberrhandle</u>

dbprocinfo

Returns information about a DBPROCESS connection.

Syntax

RETCODE dbprocinfo (PDBPROCESS *pdbproc*, **LPDBPROCINFO** *pdbprocinfo*);

Arguments

pdbproc

The DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL ServerTM 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

pdbprocinfo

A pointer to a DBPROCINFO structure that DB-Library will fill with information about the specified connection.

The DBPROCINFO structure is defined as follows:

```
typedef struct
```

{

```
DBINT SizeOfStruct;
BYTE ServerType;
USHORT ServerMajor;
USHORT ServerMinor;
USHORT ServerRevision;
CHAR ServerName[MAXSERVERNAME+1];
CHAR NetLibName[MAXNETLIBNAME+1];
CHAR NetLibConnStr[MAXNETLIBCONNSTR+1];
```

} DBPROCINFO, PTR LPDBPROCINFO;

The DBPROCINFO fields are described here.

Field	Description
SizeOfStruct	Before calling dbprocinfo , set this field equal to the value returned by the C sizeof function for the DBPROCINFO structure.
ServerType	Is one of the following: SERVTYPE_MICROSOFT if you are connected to a server running SQL Server. SERVTYPE_UNKNOWN if you are connected to an unknown type of SQL Server.
ServerMajor	Is the <i>XX</i> value in the <i>XX</i> . <i>YY</i> . <i>ZZZ</i> version number of the SQL Server you are connected to. For example, 6.
ServerMinor	Is the <i>YY</i> value in the <i>XX</i> . <i>YY</i> . <i>ZZZ</i> version number of the SQL Server you are connected to. For example, 0.
ServerRevision	Is the <i>ZZZ</i> value in the <i>XX</i> . <i>YY</i> . <i>ZZZ</i> version number of the SQL Server you are connected to. For example, 101.
ServerName	Is the name of the SQL Server to which you are connected.
NetLibName	Is the name of the Net-Library DLL used to connect to SQL Server.
NetLibConnStr	Is the Net-Library connection string used to connect to SQL Server.

Returns

SUCCEED or FAIL.

Remarks

Before calling **dbprocinfo**, set the **SizeOfStruct** field equal to the value returned by the C **sizeof** function for the DBPROCINFO structure. The **dbprocinfo** function fills the supplied DBPROCINFO structure with information about the DBPROCESS connection. See Also

<u>dbopen</u>

dbprocmsghandle

Supplies a user function to handle Microsoft® SQL Server™ 2000 messages for a specific DBPROCESS connection.

Syntax

DBMSGHANDLE_PROC dbprocmsghandle (PDBHANDLE *pdbhandle*, **DBMSGHANDLE_PROC** *message_handler*)

Arguments

pdbhandle

Either a DBPROCESS structure or a LOGINREC structure. If this is a DBPROCESS connection, the *message_handler* will be used for that connection. If this is a LOGINREC structure, the *message_handler* will automatically be used for all future DBPROCESS connections opened (with **dbopen**) using the LOGINREC.

message_handler

A pointer to the user function that is called whenever DB-Library receives an error or informational message from SQL Server for the existing or future connection specified in *pdbhandle*. Define the connection message-handler function in exactly the same way as the application message-handler function specified in **dbmsghandle**.

Returns

If *pdbhandle* is a DBPROCESS, a pointer to the previously installed connection message handler is returned. This can be NULL.

If *pdbhandle* is a LOGINREC, a pointer to the newly installed connection message handler is returned, or NULL if this function fails.

Remarks

This function is similar to the **dbmsghandle** function. While **dbmsghandle** installs a message handler global to the entire DB-Library application, **dbprocmsghandle** installs a message handler for a specific DBPROCESS connection.

When DB-Library receives a message from SQL Server using a connection that has a connection message handler (installed using **dbprocmsghandle**), only the connection message handler is called. The application message handler is not called. Because a connection message handler is associated with a connection and is not global to the entire DB-Library application, the connection message-handler code does not need to protect against reentrancy by DB-Library.

Note that this function is not supported for Microsoft MS-DOS®.

See Also

<u>dbmsghandle</u>

dbprrow

Prints all rows returned from Microsoft® SQL Server[™] 2000.

Syntax

RETCODE dbprrow (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL.

Remarks

This function displays the rows for a set of query results on the default output device in a format similar to that used by Isql.exe. This function reads and prints all the rows. Although it doesn't require allocating program variables to store the data and calling **dbbind**, the format is predetermined.

This function can be called after **dbresults** has succeeded.

When using this function, do not call **dbnextrow** to loop through the rows.

The **dbprrow** function is useful for debugging. The output line width is set at 80 columns.

This function is not supported with the Microsoft Windows® operating system.

See Also

<u>dbbind</u>

<u>dbprhead</u>

<u>dbnextrow</u>

<u>dbresults</u>

dbprtype

Converts a Microsoft[®] SQL Server[™] 2000 token value to a readable string.

Syntax

LPCSTR dbprtype (INT token);

Arguments

token

Is the SQL Server token value.

Returns

A pointer to a null-terminated string that is the readable translation of the SQL Server token value. The pointer points to space that is never overwritten, so it is safe to call this function more than once in the same statement. If the *token* is unknown, the function returns a pointer to an empty string. The strings correspond to SQL Server data type names.

Remarks

Functions such as **dbcoltype** and **dbalttype** return SQL Server token values. To print out what the token value means, use **dbprtype**.

The following token values are used by **dbprtype**.

Token value	Data type
SQLINT1	tinyint
SQLINT2	smallint
SQLINT4	int
SQLMONEY	money
SLQFLT8	float
SQLDATETIME	datetime
SQLBIT	bit

SQLCHAR	char
SQLVARCHAR	varchar
SQLTEXT	text
SQLBINARY	binary
SQLVARBINARY	varbinary
SQLIMAGE	image
SQLDECIMAL	decimal
SQLNUMERIC	numeric
SQLINTN	integer-null
SQLDATETIMN	datetime-null
SQLMONEYN	money-null
SQLFLTN	float-null
SQLAOPSUM	sum
SQLAOPAVG	avg
SQLAOPCNT	count
SQLAOPMIN	min
SQLAOPMAX	max
SQLDATETIM4	smalldatetime
SQLMONEY4	smallmoney
SQLFLT4	real

See Also

<u>dbaltop</u>

DB-Library for C Data Types

<u>dbalttype</u>

New Features in Transact-SQL

<u>dbcoltype</u>

dbresults

Sets up the results of the next query.

Syntax

RETCODE dbresults (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED, FAIL, NO_MORE_RESULTS, or NO_MORE_RPC_RESULTS. The most common reason for failing is a NULL or inactive *dbproc*. NO_MORE_RPC_RESULTS is returned when stored procedure return information is available from one stored procedure in a batch of multiple stored procedures. NO_MORE_RESULTS 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 **dbdataready** before **dbresults** and set the DB-Library time-out to regain control periodically.

Remarks

This function sets up the next statement in the command batch for processing. It is called after **dbsqlexec** or **dbsqlok**. The **dbresults** function returns SUCCEED or NO_MORE_RESULTS on the first call if **dbsqlexec** or **dbsqlok** has returned SUCCEED, unless a network error or out-of-memory error has occurred. After

dbresults returns SUCCEED, the user typically processes any rows with **dbnextrow**.

The **dbresults** function must be called for each statement in the command batch, whether or not the statement returns any rows. If the application code doesn't know how many statements are in the batch, **dbresults** can be called until it returns NO_MORE_RESULTS. Ordinarily, call **dbresults** one time for any stored procedure in the command batch. However, if the stored procedure contains more than one Transact-SQL SELECT statement, call **dbresults** one time for each SELECT statement. The easiest way to do this is to continue to call **dbresults** until it returns NO_MORE_RESULTS.

You must call **dbresults** until it returns NO_MORE_RESULTS, or *any* continued use of the DBPROCESS causes the DB-Library error 10038 "Results Pending".

Examples

This example shows the typical sequence of calls when using **dbresults** with **dbsqlexec**:

```
DBINT xvariable;
DBCHAR yvariable[10];
```

```
// Read the query into the command buffer.
dbcmd(dbproc, "SELECT x = 100, y = 'hello'");
```

```
// Send the query to SQL Server.
dbsqlexec(dbproc);
// Get ready to process the results of the query.
dbresults(dbproc);
// Bind column data to program variables.
dbbind(dbproc, 1, INTBIND, (DBINT) 0, (BYTE *) &xvariable);
dbbind(dbproc, 2, STRINGBIND, (DBINT) 0, yvariable);
// Now process each row.
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
```

// C-code to print or process row data }

See Also

<u>dbbind</u>

<u>dbnextrow</u>

<u>dbcancel</u>

<u>dbsqlexec</u>

<u>dbcanquery</u>

<u>dbsqlok</u>

dbrows

Indicates whether the current statement returned rows.

Syntax

RETCODE dbrows (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL.

Remarks

Call **dbrows** after **dbresults** returns SUCCEED. Note that **dbrows** should be called before **dbnextrow**; otherwise, **dbrows** returns an incorrect value. Use **dbcmdrow** to determine whether the current statement can return rows (that is, a Transact-SQL SELECT or EXECUTE statement on a stored procedure containing a SELECT statement).

Even if **dbrows** returns FAIL, you must still process the results by calling **dbnextrow** until it returns NO_MORE_ROWS.

See Also

dbcmdrow dbresults

dbnextrow

<u>dbrowtype</u>

dbrowtype

Returns the type of the current row.

Syntax

STATUS dbrowtype (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

One of four values:

- If the current row is a regular row, REG_ROW is returned.
- If the current row is a compute row, the *computeid* of the row is returned (for more information about *computeid*, see <u>dbaltbind</u>).
- If no rows have been read, NO_MORE_ROWS is returned.
- If **dbrowtype** was unsuccessful, FAIL is returned.

Remarks

Because **dbnextrow** returns the type (regular or compute) of the current row, you can usually determine the row type without using **dbrowtype**.

See Also

<u>dbnextrow</u>

dbserverenum

Searches for the names of Microsoft® SQL Servers[™] 2000 locally, over the network, or both.

Syntax

```
INT dbserverenum (
USHORT searchmode,
LPSTR servnamebuf,
USHORT sizeservnamebuf,
LPUSHORT numentries );
```

Arguments

searchmode

Defines whether the **dbserverenum** function checks for server names locally, on the network, or both.

Set *searchmode* to LOC_SEARCH to search for the names of servers in the Win.ini file (Microsoft Windows®) or in the following subtree of the Microsoft Windows NT® 4.0 Registry:

```
HKEY_Local_Machine\
SOFTWARE\
Microsoft\
MSSQLServer\
Client\
ConnectTo
```

The LOC_SEARCH constant has the value 0x0001.

To search for the names of servers registered by the default Net-Library, set *searchmode* to NET_SEARCH. The NET_SEARCH constant has the value 0x0002.

To search both locally and on the network for server names, use a bitwise OR operation (NET_SEARCH | LOC_SEARCH) to combine the values of

LOC_SEARCH and NET_SEARCH. In this case, a server name is duplicated if the server name is defined both locally and on the network.

DB-Library first tries to load the default Net-Library set by the DSQUERY entry in the Windows NT Registry or in the .ini file (Windows). If no default entry exists, DB-Library loads the appropriate Named Pipe Net-Library.

servnamebuf

Is a pointer to a buffer that stores the server names returned by a search. When the **dbserverenum** function successfully returns a list of server names, the *servnamebuf* buffer contains the server names separated by null characters. The end of the list is designated by two consecutive null characters. Only complete server names are copied to the buffer. When a buffer is full and there are additional names that could not be copied to the buffer, **dbserverenum** returns the value MORE_DATA. Server names are returned in unsorted order; names found using the LOC_SEARCH constant are returned before those names found using the NET_SEARCH constant. Note that **dbserverenum** can return server name seven when the SQL Server is not running. Therefore, although a server name is returned in the list of servers, it does not guarantee that you can access the server.

sizeservnamebuf

Specifies the size of the buffer for the server names returned by a successful search. The buffer size limits the maximum number of names that **dbserverenum** can return in a single call.

numentries

Returns the number of server names copied to the buffer by the current call to **dbserverenum**.

Returns

The **dbserverenum** function returns one or more of the following status code constants:

ENUM_SUCCESS (0x0000)

Indicates that the search to detect server names succeeded. ENUM_SUCCESS can be returned even when no servers are detected (when numentries = 0).

MORE_DATA (0x0001)

Indicates that DB-Library has enough memory to service the **dbserverenum** request but the buffer passed to **dbserverenum** is not large enough to hold all the names returned. Increasing the size of *sizeservnamebuf* enables the buffer to hold more server names.

NET_NOT_AVAIL (0x0002)

Indicates that the Net-Library DLL is unavailable. A Net-Library DLL might be unavailable because no DLL is found or because the available Net-Library DLL does not support the **dbserverenum** function. When the *searchmode* parameter includes the NET_SEARCH mode, **dbserverenum** calls the default Net-Library DLL. When *searchmode* includes both the NET_SEARCH and LOC_SEARCH modes (NET_SEARCH | LOC_SEARCH), **dbserverenum** can return both MORE_DATA and NET_NOT_AVAIL (MORE_DATA | NET_NOT_AVAIL).

```
OUT_OF_MEMORY (0x0004)
```

Indicates that DB-Library cannot allocate enough memory to service the **dbserverenum** request.

```
NOT_SUPPORTED (0x0008)
```

Indicates that the capability is not supported by the current Net-Library version or the current server platform.

```
ENUM_INVALID_PARAM (0x0010)
```

Indicates that an invalid parameter, such as a null pointer, was passed to the function.

Remarks

Using **dbserverenum**, you can obtain the names of servers to which you can connect. The **dbserverenum** function can search for the names of SQL Servers either locally or over the network. For network searches, the Net-Library DLL used must support the **dbserverenum** function. A Net-Library DLL registers and finds servers on networks for a specific network environment such as LAN

Manager - based networks.

When the search mode is set to NET_SEARCH, **dbserverenum** calls the default Net-Library DLL (for example, Named Pipes or Novell IPX/SPX). Because only one Net-Library DLL can be the default, those servers discernible to the specific Net-Library are enumerated using the NET_SEARCH parameter. You must add a specific server entry in the Windows NT 4.0 Registry to use a Net-Library DLL other than the default Net-Library DLL. Servers listed in the Registry are enumerated when the search mode is set to LOC_SEARCH.

See Also

<u>dbopen</u>
dbsetavail

Marks a DBPROCESS as being available for general use.

Syntax

void dbsetavail (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Remarks

Any subsequent call to **dbisavail** returns TRUE until some use is made of DBPROCESS. Many DB-Library functions automatically set DBPROCESS to "not available". This is useful when many different parts of a program are attempting to share a single DBPROCESS structure.

See Also

<u>dbisavail</u>

dbsetlapp

Sets the application name in the LOGINREC structure.

Syntax

RETCODE DBSETLAPP (PLOGINREC loginrec, **LPCSTR** application);

Arguments

loginrec

Is a pointer to a LOGINREC structure, which is passed as a parameter to **dbopen**. You can get one of these pointers by calling **dblogin**.

application

Is the application name that will be sent to Microsoft® SQL Server[™] 2000. It must be a null-terminated character string. The maximum length of the string is 30 characters, not including the null-terminating character. SQL Server stores only the first 16 characters and ignores the rest.

Returns

SUCCEED or FAIL.

Remarks

DBSETLAPP sets the application field in the LOGINREC structure. For it to have any effect, it must be called before **dbopen**. **DBSETLAPP** is optional. If you do not call DBSETLAPP, the *application* parameter of the LOGINREC structure is set to the calling application name by default instead of to NULL, as it is in SQL Server version 6.0 or earlier.

SQL Server uses the application name in its **sysprocesses** table to help identify your process. If you set the application name, you see it if you query the **sysprocesses** table in the **master** database.

See Also

<u>dblogin</u>

<u>dbsetlpwd</u>

<u>dbopen</u>

<u>dbsetluser</u>

<u>dbsetlhost</u>

dbsetlfallback

Indicates that **dbopen** can connect to a failover server if an attempt to connect to a primary server fails.

Syntax

RETCODE DBSETLFALLBACK (PLOGINREC *ploginrec,* **LPCSTR** *pstatus* **);**

Arguments

ploginrec

Is a pointer to a LOGINREC structure, which is passed as a parameter to **dbopen**.

pstatus

Specifies whether failover support is enabled or not. Set *pstatus* to "ON" to enable failover support. Set *pstatus* to "OFF" to disable failover support.

Returns

SUCCEED or FAIL.

Remarks

This function only works with a standby server configuration; it does not apply to a virtual server cluster/failover configration. Call the **DBSETLFALLBACK** function to indicate that a failover server can be used by a subsequent call to **dbopen**. When successfully connected to the primary server, the **dbopen** function automatically determines the current failover server and verifies that the failover information is stored in the registry. You must call the **DBSETLFALLBACK** function before you call the **dbopen** function. If the attempt to connect to the primary server fails, **dbopen** attempts to connect to the failover server.

Setting *pstatus* to "ON" or "OFF" determines whether failover support is

available during subsequent calls to **dbopen**.

For **DBSETLFALLBACK** to function properly, the connection time-out set by calling **dbsetlogintime** or **DBSETLTIME** must be a value greater than 0.

Examples

In this example, **DBSETLFALLBACK** specifies that failover support is enabled prior to a call to **dbopen**.

DBSETLFALLBACK (loginrec, "ON"); dbproc = dbopen (loginrec, "my_server");

dbsetlhost

Sets the workstation name in the LOGINREC structure.

Syntax

RETCODE DBSETLHOST (PLOGINREC *loginrec*, **LPCSTR** *workstation*);

Arguments

loginrec

Is a pointer to a LOGINREC structure, which is passed as a parameter to **dbopen**. You can get one of these pointers by calling **dblogin**.

workstation

Is the workstation name to be sent to Microsoft® SQL Server[™] 2000. It must be a null-terminated character string. The maximum length of the string is 30 characters, not including the null-terminating character. SQL Server stores only the first 10 characters and ignores the rest.

Returns

SUCCEED or FAIL.

Remarks

For **DBSETLHOST** to have any effect, call it before **dbopen**.

The workstation name shows up in the **sysprocesses** table in the **master** database, or on the screen if you issue an **sp_who** command.

If you do not call **DBSETLHOST**, the *workstation* parameter of the LOGINREC structure is set to the host workstation name by default instead of to NULL, as it is in SQL Server version 6.0 or earlier.

See Also

<u>dblogin</u>

<u>dbsetlpwd</u>

<u>dbopen</u>

<u>dbsetluser</u>

<u>dbsetlapp</u>

dbsetInatlang

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

Syntax

RETCODE DBSETLNATLANG (PLOGINREC loginrec, LPCSTR language);

Arguments

loginrec

Is a pointer to a LOGINREC structure to be passed as a parameter to **dbopen**. Execute **dblogin** to get LOGINREC structures.

language

Is the name of the national language to use. This parameter must be a null-terminated character string.

Returns

SUCCEED or FAIL.

Remarks

To set a language other than the Microsoft® SQL Server[™] 2000 default, call **DBSETLNATLANG** before **dbopen**. If language support is installed in the server, error messages are returned in the designated national language.

See Also

<u>dblogin</u>

<u>dbopen</u>

dbsetlogintime

Sets the number of seconds that DB-Library waits for Microsoft® SQL Server[™] 2000 to respond to a request for a DBPROCESS connection (after calling **dbopen**).

Syntax

RETCODE dbsetlogintime (INT seconds);

Arguments

seconds

Is the time-out value, or the number of seconds that DB-Library waits for a login response before timing out. A time-out value of 0 represents an infinite time-out period. The default time-out value is 60 seconds.

Returns

SUCCEED or FAIL.

See Also

<u>dberrhandle</u>

<u>dbsettime</u>

<u>dbopen</u>

dbsetlpacket

Sets the tabular data stream (TDS) packet size in an application's LOGINREC structure.

Syntax

RETCODE DBSETLPACKET (PLOGINREC loginrec, **USHORT** packet_size);

Arguments

loginrec

Is a pointer to the LOGINREC structure to be passed as an argument to **dbopen**.

packet_size

Is 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 or FAIL.

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 Microsoft® SQL Server[™] 2000. 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, because it results in fewer network reads and writes. For large data transfers, a packet size between 4092 and 8192 bytes is usually best. Any larger size can degrade performance.

The only way an application can change the TDS packet size is by using

DBSETLPACKET. If **DBSETLPACKET** is not called, all DBPROCESS 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 DBPROCESS connection to be equal to or less than the value of the *packet_size* parameter. If the server is experiencing space constraints, the packet size is set to a value less than the value of the *packet_size* parameter. Otherwise, the packet size is equal to the value of this parameter. To determine the packet size that the server has set, call **dbgetpacket**.

Different DBPROCESS connections in an application can use different packet sizes. To set different packet sizes for DBPROCESS connections, an application can either change the packet size in a single LOGINREC structure between the **dbopen** calls that create the DBPROCESS connections, or it can set different packet sizes in multiple LOGINREC structures and use these different LOGINREC structures when creating the DBPROCESS connections.

See Also

dbgetpacket dbopen dblogin

dbsetlpwd

Sets the user Microsoft® SQL Server[™] 2000 password in the LOGINREC structure.

Syntax

RETCODE DBSETLPWD (PLOGINREC loginrec, **LPCSTR** password);

Arguments

loginrec

Is a pointer to a LOGINREC structure, which is passed as a parameter to **dbopen**. To get one of these pointers, call **dblogin**.

password

Is the SQL Server password that is sent to SQL Server. It must be a null-terminated character string. The maximum length of the string is 30 characters, not including the null-terminating character.

Returns

SUCCEED or FAIL.

Remarks

For **DBSETLPWD** to have any effect, call it before **dbopen**.

You do not need to call it if the password is a null value.

See Also

<u>dblogin</u>

<u>dbsetlhost</u>

<u>dbopen</u> <u>dbsetluser</u>

<u>dbsetlapp</u>

dbsetlsecure

Sets the secure connection flag in a LOGINREC structure.

Syntax

RETCODE DBSETLSECURE (PLOGINREC ploginrec);

Arguments

ploginrec

A pointer to a LOGINREC structure, which is passed as a parameter to **dbopen**. You can get one of these pointers by calling **dblogin**.

Returns

SUCCEED or FAIL.

Remarks

By setting the secure connection flag in a LOGINREC structure with **DBSETLSECURE**, the application requests a secure, or *trusted*, connection to Microsoft® SQL Server[™] 2000. This means that SQL Server will use Windows Authentication security to establish connections made (using **dbopen**) with this LOGINREC, regardless of the current login security mode at the server. Any login ID or password supplied by **DBSETLUSER** or **DBSETLPWD** is ignored.

To use **DBSETLSECURE** and trusted connections, you must first use **xp_grantlogin** to grant SQL Server system administrator or user privilege to the appropriate Microsoft Windows NT® 4.0-based groups or users. Use **xp_revokelogin** to revoke SQL Server permissions and stop a user or group from using a trusted connection.

Note that **DBSETLSECURE** enables trusted connections even when the server is in Mixed Mode.

See Also

dblogin dbsetlhost dbopen dbsetluser dbsetlapp <u>Authentication Modes</u>

dbsetltime

Sets the connection-specific connection time-out in a LOGINREC structure.

Syntax

RETCODE DBSETLTIME (PLOGINREC ploginrec, **DWORD** seconds);

Arguments

ploginrec

A pointer to a LOGINREC structure, which is passed as a parameter to **dbopen**. You can get one of these pointers by calling **dblogin**.

seconds

The number of seconds that DB-Library waits for a login response from Microsoft® SQL ServerTM 2000 before timing out. The following special values are also available.

seconds	Description
TIMEOUT_IGNORE	Use the global connection time-out set using
	dbsetlogintime.
TIMEOUT_INFINITE	No time-out.
TIMEOUT_MAXIMUM	Maximum DB-Library connection time-out.

Returns

SUCCEED or FAIL.

Remarks

The *seconds* value overrides the global time-out value set using **dbsetlogintime**

or the default login time-out.

See Also

<u>dblogin</u>

<u>dbsetlhost</u>

<u>dbopen</u>

<u>dbsetlogintime</u>

<u>dbsetlapp</u>

<u>dbsetluser</u>

dbsetluser

Sets the username in the LOGINREC structure.

Syntax

RETCODE DBSETLUSER (PLOGINREC loginrec, **LPCSTR** username);

Arguments

loginrec

Is a pointer to a LOGINREC structure, which is passed as a parameter to **dbopen**. To get one of these pointers, call **dblogin**.

username

Is the username that is sent to Microsoft® SQL Server[™] 2000. It must be a null-terminated character string. The maximum length of the string is 30 characters, not including the null-terminating character. SQL Server uses it to determine who is attempting the connection. The SQL Server usernames are defined in the **syslogins** table in the **master** database.

Returns

SUCCEED or FAIL.

Remarks

For **DBSETLUSER** to have any effect, it must be called before **dbopen**.

See Also

<u>dblogin</u>

<u>dbsetlhost</u>

dbopen dbsetlpwd dbsetlapp

dbsetlversion

Sets the DB-Library client behavior to version 4.2 or 6.0 and later behavior in a LOGINREC structure.

Syntax

RETCODE DBSETLVERSION (PLOGINREC ploginrec, **BYTE** version)

Arguments

ploginrec

A pointer to a LOGINREC structure, which is passed as a parameter to **dbopen**. You can get one of these pointers by calling **dblogin**.

version

The DB-Library client behavior to use. Must be either DBVER60 to set DB-Library 6.0 behavior or DBVER42 to set DB-Library 4.2 behavior.

Returns

SUCCEED or FAIL.

Remarks

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

Using the DBVER60 value means that Microsoft® SQL Server[™] 2000 treats the connection as a DB-Library 6.0 client in every way. SQL Server:

- Returns **decimal** and **numeric** data values.
- Returns complete column information (including identity column information) to DB-Library and **dbcolinfo**.

Using the DBVER42 value (or not calling **DBSETLVERSION** for the LOGINREC) means that SQL Server treats the connection as a DB-Library 4.2 client. SQL Server:

- Converts **decimal** and **numeric** data values to **float** before returning them to the client.
- Returns limited, version 4.2 column information (not including identity column information) to DB-Library and **dbcolinfo**.

Note that using DBVER60 is not required to use SQL Server version 6.0 and later server cursors.

See Also

<u>dblogin</u>

<u>dbsetlhost</u>

<u>dbopen</u>

<u>dbsetlpwd</u>

<u>dbsetlapp</u>

dbsetmaxprocs

Sets the maximum number of simultaneously open DBPROCESS structures.

Syntax

RETCODE dbsetmaxprocs (SHORT *maxprocs*);

Arguments

maxprocs

Is the new limit on simultaneously open DBPROCESS structures for this particular application.

Returns

SUCCEED if the function call is successful, or else FAIL (that is, *maxprocs* <= 0).

Remarks

A DB-Library application has a maximum number of simultaneously open DBPROCESS structures. Although the default number is 25, the application can change it by calling **dbsetmaxprocs**.

Similarly, the program can find out what the current limit is by calling **dbgetmaxprocs**.

The following table shows platform-dependent connection limitations:

	Maximum number of connections per
Operating system	workstation
Microsoft® Windows NT® 4.0	Limited by amount of memory available.
Microsoft Windows® 95 or Windows 98	Limited by amount of memory available.
Windows	45 for all Microsoft® SQL Server™ 2000-running

	applications.
Microsoft MS-DOS®	15.

See Also

<u>dbgetmaxprocs</u>

<u>dbopen</u>
dbsetnull

Defines substitution values for use when binding null values.

Syntax

RETCODE dbsetnull (PDBPROCESS dbproc, INT bindtype, INT bindlen, LPCBYTE bindval);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

bindtype

Is the type of variable binding to which the substitute value applies. (For more information about the different *bindtypes*, see <u>dbbind</u>.)

bindlen

Is the length, in bytes, of the substitute value you are supplying. It is ignored in all cases except CHARBIND and BINARYBIND. All the other types are either fixed length or have a special terminator or embedded byte count that provides the length of the data.

bindval

Is a generic BYTE pointer to the value you want to use as a null value substitution. The **dbsetnull** function always makes a copy of this data, so you can free this pointer any time after this call.

Returns

SUCCEED or FAIL. The **dbsetnull** function fails if you give it a null *bindval* or if the length is smaller than 1 when CHARBIND and BINARYBIND types are used.

Remarks

The **dbbind** and **dbaltbind** functions are used to bind returned SQL Server column values to your program variables. Because a null value can be returned, there is a mechanism for defining what values should be substituted for the null value when doing automatic copying of column data to program variables. Associated with each DBPROCESS is a list of substitute values for each of the binding types. The default substitution values are as follows:

Binding type	Default substitution value
TINYBIND	0
SMALLBIND	0
INTBIND	0
BITBIND	0
CHARBIND	Empty string (padded with blanks)
STRINGBIND	Empty string (padded with blanks, null- terminated)
NTBSTRINGBIND	Empty string (null-terminated)
VARYCHARBIND	Empty string
BINARYBIND	Empty array (padded with zeros)
VARYBINBIND	Empty array
DATETIMEBIND	8 bytes of zeros
MONEYBIND	\$0.00
FLT8BIND	0.0
FLT4BIND	0.0
SMALLMONEYBIND	\$0.00
SMALLDATETIBIND	4 bytes of zeros
DECIMALBIND	0.0
NUMERICBIND	0.0

SRCDECIMALBIND	0.0
SRCNUMERICBIND	0.0

Call **dbsetnull** to supply your own null substitution values. Whenever you call **dbsetnull** to change a particular null substitution value, the new value remains in force for that DBPROCESS until you change it with another call to **dbsetnull**.

The **dbconvert** function also uses the current null substitution values when it needs to set a destination variable to null.

See Also dbaltbind dbconvert

<u>dbbind</u>

DB-Library for C Data Types

dbsetopt

Sets a Microsoft[®] SQL Server[™] 2000 or DB-Library option.

Syntax

RETCODE dbsetopt (PDBPROCESS dbproc, INT option, LPCSTR param);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server. If *dbproc* is NULL, the option is set for all active DBPROCESS structures.

option

Is the option to be set. See <u>DB-Library Options</u>.

param

Is a parameter for an option. Certain options take parameters. For example, the DBOFFSET option takes as its parameter the construct for which offsets are to be returned:

dbsetopt(dbproc, DBOFFSET, "compute")

The DBBUFFER option takes as its parameter the number of rows to be buffered. A parameter with a negative value selects the default (currently 100 rows). A value of 1 is invalid.

dbsetopt(dbproc, DBBUFFER, "500")

Note The *param* variable must always be a character string enclosed in quotation marks, even in the case of a numeric value, as in the preceding DBBUFFER example.

If the option takes no parameters, *param* is ignored and may be NULL.

Returns

SUCCEED or FAIL.

This function fails if *param* is invalid for one of the DB-Library options. However, an invalid *param* for an SQL Server option does not cause **dbsetopt** to fail, because such a parameter does not get validated until the command buffer is sent to SQL Server.

Remarks

Although SQL Server options can be set and cleared directly through Transact-SQL, the application should use **dbsetopt** and **dbclropt** to set and clear options. This provides a uniform interface for setting both SQL Server and DB-Library options. It also allows the application to use the **dbisopt** function to check the status of an option.

This function does not immediately set the options specified. They are not set until the command buffer is sent to SQL Server (by invoking the **dbsqlexec** function). An additional result for each option set is returned through the **dbresults** function.

See Also
<u>dbclropt</u>
<u>dbresults</u>
<u>dbisopt</u>
<u>dbsqlexec</u>
DB-Library Options
Bulk-Copy Functions

dbsettime

Sets the number of seconds that DB-Library waits for Microsoft® SQL ServerTM 2000 to respond during calls to **dbsqlexec**, **dbsqlok**, **dbresults**, **dbnextrow**, and **dbrpcexec**.

Syntax

RETCODE dbsettime (INT seconds);

Arguments

seconds

Is the time-out value, or the number of seconds that DB-Library waits for SQL Server to respond before timing out. The default time-out value of 0 represents an infinite time-out period.

Returns

SUCCEED or FAIL.

Remarks

This function sets the length of time, in seconds, that DB-Library waits for a SQL Server response during calls to **dbsqlexec**, **dbsqlok**, **dbresults**, **dbnextrow**, and **dbrpcexec**. The **dbsettime** function does not override existing network time-out settings.

The **dbsettime** function can be called at any time during the application, before or after a call to **dbopen**. It takes effect immediately upon being called.

To set a time-out value for calls to **dbopen**, use **dbsetlogintime**.

Note that if an application sends a command to SQL Server using **dbsqlexec**, control is not returned to the calling application until SQL Server completes the processing of the command. If the application program needs to continue execution while SQL Server is processing the command, it should send the command with **dbsqlsend**, continue its processing, and then, when it is ready to

retrieve the results, call **dbsqlok**.

The program can call **dbgettime** for the current time-out value.

See Also

<u>dbdataready</u> <u>dbrpcexec</u>

<u>dberrhandle</u>

<u>dbsetlogintime</u>

<u>dbgettime</u>

<u>dbsqlexec</u>

<u>dbnextrow</u>

<u>dbsqlok</u>

<u>dbresults</u>

<u>dbsqlsend</u>

dbsetuserdata

Saves, in a DBPROCESS structure, a pointer to user-allocated data.

Syntax

void dbsetuserdata (PDBPROCESS dbproc, LPVOID ptr);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

ptr

Is a generic BYTE pointer to the user's private data space.

Remarks

The **dbsetuserdata** function lets an application associate user data with a particular DBPROCESS, eliminating the need for global variables. This function is especially useful when an application has more than one DBPROCESS.

Make sure each application allocates the data that *ptr* points to. DB-Library never manipulates this data; it merely saves the pointer so the application can use it later.

Examples

The following example shows how **dbsetuserdata** handles deadlock, an occasional occurrence in high-volume applications. This program fragment sends updates to the SQL Server and reruns the transaction when its message handler detects deadlock:

// Deadlock detection:

// In the DBPROCESS structure, save a pointer to a DBBOOL variable // The message handler sets the variable when deadlock occurs. // The result processing logic checks the variable and resends the // transaction in case of deadlock.

// Allocate the space for the DBBOOL variable and save it in
// the DBPROCESS structure.

dbsetuserdata(dbproc, malloc(sizeof(DBBOOL)));

// Initialize the variable to FALSE.
*((DBBOOL *) dbgetuserdata(dbproc)) = FALSE;
// Run queries and check for deadlock.
deadlock:

// Did the application get here via deadlock? // If so, the server has already canceled the transaction. // Start the application again. In a real application, // the deadlock handling may need to be somewhat more // sophisticated. For instance, you may want to keep a // counter and retry the transaction a fixed number // of times.

```
if (*((DBBOOL *) dbgetuserdata(dbproc)) == TRUE)
{
    // Reset the variable to FALSE.
    *((DBBOOL *) dbgetuserdata(dbproc)) = FALSE;
}
```

```
// Start the transaction.
dbcmd(dbproc, "begin transaction ");
```

// Run the first UPDATE command.

```
dbcmd(dbproc, "UPDATE ...");
dbsqlexec(dbproc);
while (dbresults(dbproc) ! = NO_MORE_RESULTS)
{
// application code
}
// Did the application deadlock?
if (*(DBBOOL *) dbgetuserdata(dbproc)) == TRUE)
goto deadlock;
// Run the second UPDATE command.
dbcmd(dbproc, "UPDATE ...");
dbsqlexec(dbproc);
while (dbresults(dbproc) ! = NO_MORE_RESULTS
{
// application code
}
```

```
// Did the application deadlock?
if (*((DBBOOL *) dbgetuserdata(dbproc)) == TRUE)
goto deadlock;
```

```
// No deadlock -- Commit the transaction.
dbcmd(dbproc, "COMMIT TRANSACTION");
dbsqlexec(dbproc);
dbresults(dbproc);
```

// SERVERMSGS

// This is the server message handler. Assume that the dbmsghandle
// function installed it earlier in the application.

servermsgs(dbproc, msgno, msgstate, severity, msgtext, srvname, proc

```
DBPROCESS *dbproc;
DBINT
          msgno;
int
       msgstate;
       severity;
int
char
char
         *msgtext;
         srvname;
char *procname;
DBUSMALLINT line;
{
// Is this a deadlock message?
if (msgno = 1205)
{
 // Set the deadlock indicator.
  * ((DBBOOL *) dbgetuserdata(dbproc)) = TRUE;
 return (O);
}
```

```
// Normal message handling code here.
}
```

See Also

<u>dbgetuserdata</u>

dbsqlexec

Sends a command batch to Microsoft[®] SQL Server[™] 2000.

Syntax

RETCODE dbsqlexec (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL. The most common reason for failing is a Transact-SQL syntax error or SQL Server permission violation. Other reasons include incorrect column or table names. The **dbsqlexec** function also fails if previous results were not processed, if no statement was specified, or if a network connection is broken.

Note that if a series of commands is sent to SQL Server and if one or more of the commands contains syntax errors, SQL Server processes none of the commands, and **dbsqlexec** returns FAIL.

Remarks

This function sends Transact-SQL statements, stored in the command buffer of the DBPROCESS, to SQL Server. Statements can be added to the DBPROCESS structure by calling **dbcmd** or **dbfcmd**.

The **dbsqlexec** function is equivalent to **dbsqlsend** followed by **dbsqlok**. However, after sending a query to SQL Server, **dbsqlexec** waits until it receives a response or until the time-out period has elapsed. If you prefer not to have your application wait, substitute **dbsqlsend**, **dbdataready**, and **dbsqlok**.

After **dbsqlexec**, the application must call **dbresults** to process the results.

Example

This example shows the typical sequence of calls:

```
DBINT xvariable;
DBCHAR yvariable[10];
```

```
// Place the query into the command buffer.
dbcmd(dbproc, "SELECT x = 100, y = 'hello'");
```

```
// Send the command buffer to SQL Server.
dbsqlexec(dbproc);
```

```
// Get ready to process the results of the query.
dbresults(dbproc);
```

```
// Bind column data to program variables.
dbbind(dbproc, 1, INTBIND, (DBINT) 0, (BYTE *) &xvariable);
dbbind(dbproc, 2, STRINGBIND, (DBINT) 0, yvariable);
```

```
// Now process each row.
while (dbnextrow(dbproc) != NO_MORE_ROWS)
{
    // C-code to print or process row data
}
```

See Also

<u>dbcmd</u> <u>dbsettime</u> dbfcmd <u>dbsqlok</u>

<u>dbnextrow</u>

<u>dbsqlsend</u>

<u>dbresults</u>

dbsqlok

Verifies the correctness of a command batch.

Syntax

RETCODE dbsqlok (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL. The most common reason for failing is a Transact-SQL syntax error.

Remarks

dbsqlok following **dbsqlsend** is the equivalent of **dbsqlexec**. This function must be called after **dbsqlsend** returns SUCCEED. When **dbsqlok** returns, then **dbresults** can be called to process the results. In a batch, if the last statement fails, then **dbsqlok** returns FAIL but there may still be some result set to process. So even if **dbsqlok** returns FAIL, dbresults should be called to process any outstanding result set (until NO_MORE_RESULTS is returned).

The **dbsqlok** function is also useful in text-update operations. When chunks of text are sent to SQL Server using **dbmoretext**, **dbsqlok** must be called before the first call to **dbmoretext** and after the last call to **dbmoretext**. For an example of its use in this context, see <u>dbwritetext</u>.

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 **dbdataready** before **dbsqlok** and set the DB-Library time-out to regain control periodically.

See Also

dbcmd dbsettime dbfcmd dbsqlexec dbmoretext dbsqlsend dbnextrow dbwritetext

<u>dbresults</u>

dbsqlsend

Sends a command batch to Microsoft® SQL Server[™] 2000 and does not wait for a response.

Syntax

RETCODE dbsqlsend (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation/ SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL.

Remarks

This function sends Transact-SQL statements, stored in the command buffer of the DBPROCESS, to SQL Server. Statements can be added to the command buffer by calling **dbcmd** or **dbfcmd**.

After **dbsqlsend** returns SUCCEED, **dbsqlok** must be called to verify the accuracy of the command batch. Then **dbresults** can be called to process the results.

This function is particularly useful for multitasking with the Microsoft Windows® operating system.

See Also

Building Applications dbresults <u>dbcmd</u>

<u>dbsettime</u>

<u>dbfcmd</u>

<u>dbsqlexec</u>

<u>dbnextrow</u>

<u>dbsqlok</u>

dbstrcpy

Copies a portion of the command buffer in the DBPROCESS structure to a specified memory location.

Syntax

RETCODE dbstrcpy (PDBPROCESS dbproc, INT start, INT numbytes, LPSTR dest);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

start

Is the character position in the command buffer from which to start copying. The first character position is 0. If *start* is greater than the length of the command buffer, a null terminator is inserted at *dest*[0].

numbytes

Is the number of characters to copy.

numbytes setting	Result
< 0	dbstrcpy copies the entire command buffer. It is
	valid to copy 0 bytes.
= 0	A null terminator is inserted at <i>dest</i> [0].
> 0	dbstrcpy copies the number of bytes available and
	returns SUCCEED.

dest

Is a pointer to the memory location to copy the source string into.

Returns

SUCCEED or FAIL. FAIL is returned if *start* is a negative number.

Remarks

The copy is null-terminated. The **dbstrcpy** function assumes that the destination string is large enough to receive the source string. If not, unexpected errors may occur.

Internally, the command buffer is a linked list of nonterminated text strings. Parts of the command buffer can be located and copied using the **dbgetchar**, **dbstrcpy**, and **dbstrlen** functions.

See Also dbcmd dbgetchar dbfcmd dbstrlen dbfreebuf

dbstrlen

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

Syntax

INT dbstrlen (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The length, in characters, of the command buffer.

Remarks

Internally, the command buffer is a linked list of nonterminated text strings. Parts of the command buffer can be located and copied using the **dbgetchar**, **dbstrcpy**, and **dbstrlen** functions.

See Also dbcmd dbgetchar dbfcmd dbstrcpy dbfreebuf

dbuse

Uses a particular database.

Syntax

RETCODE dbuse (PDBPROCESS dbproc, LPCSTR dbname);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

dbname

Is a character pointer to a null-terminated string specifying the database name.

Returns

SUCCEED or FAIL.

Remarks

This function executes a Transact-SQL USE statement for the specified database for a particular DBPROCESS structure. It sets up the statement and calls **dbsqlexec** and **dbresults**.

If the USE statement fails because the requested database has not yet completed a recovery process, **dbuse** continues to send USE statements at 1-second intervals until it either succeeds or encounters some other error.

The function uses the *dbproc* provided by the caller and its command buffer.

Any existing statements in the buffer are lost, and the command buffer is cleared by **dbuse** when it is finished.

See Also

<u>dbchange</u> <u>dbresults</u>

<u>dbname</u>

<u>dbsqlexec</u>

dbvarylen

Determines whether the specified data in a regular result column can vary in length.

Syntax

BOOL dbvarylen (PDBPROCESS dbproc, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the regular result column of interest. The first column is number 1. For further information, see <u>dbadata</u>.

Returns

TRUE or FALSE, indicating whether or not the column's data can vary in length. **dbvarylen** also returns FALSE if the column number is out of range.

This function returns TRUE if:

- The result column is derived from a database column of type **varchar**, **varbinary**, **text**, or **image**.
- The source database column is defined as NULL, meaning that it may contain a null value.

Remarks

Use **dbvarylen** especially with programs that handle ad hoc queries and that need to be alerted to possible null data or data of variable length. Call **dbvarylen** only after **dbresults** returns SUCCEED.

To get a column's data type, execute **dbcoltype**.

See Also
<u>dbcollen</u>
<u>dbdatlen</u>
<u>dbcolname</u>
DB-Library for C Data Types
<u>dbcoltype</u>
<u>dbnumcols</u>
<u>dbdata</u>
<u>dbprtype</u>
dbwillconvert

Determines whether a specific data type conversion is available within DB-Library.

Syntax

BOOL dbwillconvert (INT srctype, INT desttype);

Arguments

srctype

Is the data type of the data to be converted. This parameter can be any of the Microsoft® SQL ServerTM 2000 data types.

desttype

Is the data type into which the source data is to be converted. This parameter can be any of the SQL Server data types.

Returns

TRUE if the data type conversion is supported; FALSE otherwise.

Remarks

When **dbconvert** is asked to perform a conversion that it doesn't support, it calls a user-supplied error handler (if any) and returns -1.

The **dbconvert** function can convert data stored in any of the SQL Server data types (not all conversions are allowable):

SQL Server data type	Program variable type
SQLCHAR	DBCHAR
SQLTEXT	DBCHAR

SQLBINARY	DBBINARY
SQLIMAGE	DBBINARY
SQLINT1	DBTINYINT
SQLINT2	DBSMALLINT
SQLINT4	DBINT
SQLFLT8	DBFLT8
SQLBIT	DBBIT
SQLMONEY	DBMONEY
SQLDATETIME	DBDATETIME
SQLDATETIM4	DBDATETIM4
SQLMONEY4	DBMONEY4
SQLFLT4	DBFLT4
SQLDECIMAL	DBDECIMAL
SQLNUMERIC	DBNUMERIC

For more information about data type conversions, see <u>dbconvert</u>.

See Also

<u>dbaltbind</u>

<u>dbconvert</u>

<u>dbbind</u>

DB-Library Options

dbwinexit (Windows only)

Informs DB-Library that the Microsoft® Windows® application is about to exit.

Syntax

void dbwinexit (void);

Remarks

DB-Library for Windows maintains information about each application that has referenced it. DB-Library creates the information when a library application calls **dbinit**; it does this to prevent conflicts between applications that use DB-Library concurrently. For DB-Library to release this information, the application must call **dbwinexit** just before it exits. You should put the call to **dbwinexit** within the message handling code for the WM_DESTROY message. For example:

```
case WM_DESTROY:
dbwinexit();
break;
```

This call releases the memory DB-Library allocated to keep track of this application and makes that memory available to other applications.

It should be noted that **dbwinexit** does not close any connections. Use **dbclose** or **dbexit** to close connections.

IMPORTANT After your application has called **dbwinexit**, it cannot call any other DB-Library function. If you have called **dbwinexit** and then need to issue one or more DB-Library calls, you must call **dbinit** again to reregister your application.

See Also

<u>dbinit</u>

Cursor Functions

DB-Library for C version 7.0 supports several cursor functions.

dbcursor

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

RETCODE dbcursor (PDBCURSOR hc, INT optype, INT row, LPCSTR table, LPCSTR values);

Arguments

hc

Is the cursor handle previously returned by **dbcursoropen**.

optype

Specifies the type of cursor operation to perform on a row or rows in the fetch buffer, as follows.

optype	Description
CRS_DELETE	Deletes row(s).
CRS_INSERT	Inserts a single row using data specified in <i>values</i> .
CRS_LOCKCC	Locks row(s).
	Client cursor: 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.

	Transparent server cursor, explicit server cursor:
	An update intent lock is placed on the data page that
	contains the specified <i>row</i> . This lock is released when the
	next fetch is performed or when the cursor is closed.
CRS_REFRESH	Refreshes row(s) by retrieving current row data from
	Microsoft® SQL Server™ 2000.
CRS_UPDATE	Updates row(s) using data specified in <i>values</i> .

If the cursor was opened using a *concuropt* of CUR_READONLY (specified in **dbcursoropen**), only CRS_REFRESH 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 CRS_REFRESH, 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:

- CRS_DELETE
- CRS_LOCKCC
- CRS_REFRESH

table

Is the table to which the *optype* operation applies. If *optype* is CRS_REFRESH use NULL. It must be one of the tables specified in the FROM clause of the SELECT statement (specified in **dbcursoropen**) that defines the cursor. If the FROM clause includes only one table, this

parameter is not required, and you can specify that table or NULL.

Client cursor:

If the FROM clause includes more than one table, this parameter is required unless:

- *optype* is CRS_INSERT and *values* points to a complete INSERT statement.
- *optype* is CRS_UPDATE 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 **dbcursor** 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 pointer to a string that contains a Transact-SQL statement or clause, or NULL. This parameter specifies the data to be inserted or updated. The following table lists the valid *values* parameters for each *optype*.

optype	Values
CRS_DELETE	NULL
CRS_INSERT	Can be one of the following: NULL, indicating that the single row to be inserted will come from the data stored in the bound program variables (<i>pvaraddr</i> and <i>poutlen</i> in dbcursorbind) for the specified <i>row</i> in the fetch buffer. This requires overwriting the values in the bound program variables for an existing row. A pointer to 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 pointer to 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.
CRS_LOCKCC	NULL
CRS_REFRESH	NULL
CRS_UPDATE	Can be one of the following:
	NULL, indicating that the changes made to a single row come from the data stored in the bound program variables (<i>pvaraddr</i> and <i>poutlen</i> in dbcursorbind) for the specified <i>row</i> in the fetch buffer. To do this, the SELECT statement that defines the cursor (specified in dbcursoropen) cannot include an expression (for example, "length + 10") or a function (such as CONVERT) in the select list.
	A pointer to 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 pointer to 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 or FAIL.

This function can fail for the following reasons:

• The cursor is opened as read-only, no updates allowed.

- A server or connection failure or time-out 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 **dbcursor** does not affect the current cursor position.

When *optype* is CRS_UPDATE and the *values* parameter is not NULL and points to a string that contains the UPDATE statement or SET clause, the bound program variables (*pvaraddr* and *poutlen* in **dbcursorbind**) are automatically refreshed to their newly updated values.

The select list used to define the cursor (specified in **dbcursoropen**) can contain **timestamp** or **identity** columns. When *optype* is CRS_INSERT or CRS_UPDATE and the *values* parameter is NULL (indicating that new data is obtained from bound program variables), any read-only columns (including **timestamp** or **identity** columns) in the fetch buffer are skipped if **dbsetlversion** was called with DBVER60. Because no attempt is made to change these read-only columns, the insert or update can succeed without receiving errors.

When using CRS_UPDATE, 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 **dbcursoropen**) for that row will be FTC_MISSING.

• 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 CRS_DELETE, deleted rows will be missing from a keyset cursor (later fetches will have a row status of FTC_MISSING), and will disappear from later fetches using dynamic cursors.

Client cursor:

When using CRS_INSERT with a keyset cursor, the inserted row does not appear in the cursor result set, and thus does not appear in later fetches.

Transparent server cursor, explicit server cursor:

When *optype* is one of the following:

- CRS_INSERT with a keyset cursor involving only one table
- CRS_UPDATE 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

Bulk-Copy Functions dbcursorfetch dbcursorbind dbcursorinfo dbcursorclose dbcursoropen <u>dbcursorcolinfo</u>

dbcursorbind

Binds a column of a client cursor, transparent server cursor, or explicit server cursor to an array of program variables. This array of program variables is filled with result data after every fetch or refresh.

Syntax

RETCODE dbcursorbind (PDBCURSOR hc, INT col, INT vartype, DBINT varlen, LPDBINT poutlen, LPBYTE pvaraddr);

Arguments

hc

Is the cursor handle created by **dbcursoropen**.

col

Is the column number in the cursor to be bound to a program variable. The first column is number 1.

vartype

Is the bind type used to bind the **col** column to the *pvaraddr* array of program variables. This bind type is identical to the *vartype* parameter of **dbbind** and uses the same conversion rules. The *vartype* parameter indicates the program variable data type of the *pvaraddr* array elements.

In addition to the **dbbind** bind types, *vartype* can be set to NOBIND, indicating that no binding is established for a column. After a fetch:

• Each element of the *pvaraddr* array will contain a pointer to the column data for that row, similar to **dbdata**.

• Each element of the *poutlen* array will contain the length of the column data for that row, similar to **dbdatlen**.

varlen

Is the maximum length, in bytes, of a variable-length data type. This parameter must be supplied when *vartype* is one of the following:

- CHARBIND, VARYCHARBIND
- STRINGBIND, NTBSTRINGBIND
- BINARYBIND, VARYBINBIND

This parameter is ignored for fixed-length data types (all other *vartype* values except those given earlier) and when *vartype* is NOBIND.

poutlen

Is a pointer to an array of DBINT elements, one element for each row in the fetch buffer. The required number of elements in this array is given later. After a fetch, each element of this array will contain the length of the column data for that row, similar to **dbdatlen**. A length element value of 0 indicates that the column is NULL.

If *poutlen* is set to NULL, the lengths are not returned.

Before calling **dbcursor** to update or insert a row with values from the *pvaraddr* array of bound program variables, set the corresponding *poutlen* element (for that row in the fetch buffer) to one of the following values:

- Set an element to 0 to indicate a NULL value.
- When *vartype* is NOBIND, CHARBIND, VARYCHARBIND, BINARYBIND, or VARYBINBIND, set the element to the length, in bytes, of the new column data.
- When vartype is STRINGBIND or NTBSTRINGBIND, any nonzero

value is ignored, and the length of the string is determined by scanning for the null terminator.

• For fixed-length data types, nonzero values are ignored.

Client cursor, transparent server cursor:

The number of elements in this array must equal the *nrows* parameter of **dbcursoropen**.

Explicit server cursor:

The maximum number of elements in this array is the *nrows* parameter of **dbcursoropen**. Before a fetch is made, the number of elements in this array must be equal to or greater than the *nfetchnrows* parameter that will be passed to **dbcursorfetchex**.

pvaraddr

Is a pointer to an array of elements, one element for each row in the fetch buffer. The required number of elements in this array is given later. After a fetch, each element of this array will contain:

- A copy of the column data for that row, converted as specified by the *vartype* parameter (similar to **dbbind**) when *vartype* is one of the standard **dbbind** bind types.
- A pointer to the actual column data for that row (similar to **dbdata**) when *vartype* is NOBIND.

Setting *pvaraddr* to NULL breaks any existing binding established by a previous call to **dbcursorbind** for the column.

Client cursor, transparent server cursor:

The number of elements in this array must equal the *nrows* parameter of **dbcursoropen**.

Explicit server cursor:

The maximum number of elements in this array is the *nrows* parameter of **dbcursoropen**. Before a fetch is made, the number of

elements in this array must be equal to or greater than the *nfetchnrows* parameter that will be passed to **dbcursorfetchex**.

Returns

SUCCEED or FAIL.

Remarks

If **dbcursorbind** is called more than once for any column, only the last call will be effective.

See Also

Bulk-Copy Functions dbcursorfetch dbcursor dbcursorinfo dbcursorclose dbcursoropen dbcursorcolinfo

dbcursorclose

Closes the client cursor, transparent server cursor, or explicit server cursor, and releases the memory associated with a cursor handle or connection.

Syntax

RETCODE dbcursorclose (PDBHANDLE handle);

Arguments

handle

Is a DBPROCESS connection or DBCURSOR cursor handle. Cast *handle* to a PDBHANDLE value (for example, **(PDBHANDLE)mydbproc** or **(PDBHANDLE)mycursor)** to avoid compiler warnings. The following table shows the **dbcursorclose** operation for each type of *handle*.

Handle	Operation
PDBCURSOR	Closes the specified cursor.
PDBPROCESS	Closes all open cursors using the specified connection.

Returns

SUCCEED or FAIL. If *handle* is a DBPROCESS connection, and this function returns FAIL, call **dbcursorclose** again to continue closing all open cursors for the specified connection.

Remarks

After issuing **dbcursorclose**, the cursor handle should not used.

See Also

Bulk-Copy Functions

<u>dbcursorfetch</u>

<u>dbcursor</u>

<u>dbcursorinfo</u>

<u>dbcursorbind</u>

<u>dbcursoropen</u>

<u>dbcursorcolinfo</u>

dbcursorcolinfo

Returns information about the specified column of a client cursor, transparent server cursor, or explicit server cursor.

Syntax

RETCODE dbcursorcolinfo (PDBCURSOR hcursor, INT column, LPSTR colname, LPINT coltype, LPDBINT collen, LPINT usertype);

Arguments

hcursor

Is the cursor handle created by **dbcursoropen**.

column

Is the column number in the cursor for which information is to be returned. The first column is number 1.

colname

Is a pointer to the program variable that will contain the name of the column. The program variable should be large enough to accommodate the column name. If *colname* is set to NULL or -1, the column name is not returned.

coltype

Is a pointer to the program variable that will contain the data type token of the column. If *coltype* is set to NULL or -1, the column data type is not returned.

collen

Is a pointer to the program variable that will contain the maximum length of

the column, in bytes. If *collen* is set to NULL or -1, the maximum column length is not returned.

usertype

Is a pointer to the program variable that will contain the user-defined data type of the column. If *usertype* is set to NULL or -1, the column user-defined data type is not returned.

Returns

SUCCEED or FAIL.

Remarks

Call **dbcursorcolinfo** after **dbcursoropen** returns a valid cursor handle. The **dbcolinfo** function returns more detailed information about a cursor column.

See Also

dbcolinfo dbcursorfetch dbcursor dbcursorinfo dbcursorbind

dbcursoropen

<u>dbcursorclose</u>

dbcursorfetch

Fetches a block of rows (called the fetch buffer) from a client cursor or transparent server cursor, and stores the rows in the bound program variables established using **dbcursorbind**. If you are connected to Microsoft® SQL Server[™] version 6.0 or later, you should use **dbcursorfetchex**.

Syntax

RETCODE dbcursorfetch (PDBCURSOR hc, INT fetchtype, INT rownum);

Arguments

hc

Is the cursor handle created by **dbcursoropen**.

fetchtype

Specifies the type of fetch to execute, changing the position of the fetch buffer within the cursor result set. The following table describes the different *fetchtype* values.

fetchtype	Description
FETCH_FIRST	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 result set.
FETCH_NEXT	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 FETCH_FIRST.
FETCH_PREV	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 dbcursoropen) before the first row of the current fetch buffer.
FETCH_RANDOM	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 result set.
FETCH_RELATIVE	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.
FETCH_LAST	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 result 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 **dbcursoropen**.

For a forward-only dynamic cursor (*scrollopt* is CUR_FORWARD in **dbcursoropen**), you can use only FETCH_FIRST or FETCH_NEXT.

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 FETCH_RANDOM or FETCH_RELATIVE. Specify 0 for any other *fetchtype*.

When *fetchtype* is FETCH_RANDOM, the first row of the new fetch buffer is the *rownum* row (counting forward from the beginning) of the keyset cursor result set. The *rownum* parameter must be positive.

When *fetchtype* is FETCH_RELATIVE:

- 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 **dbcursor** with *optype* set to CRS_REFRESH.

Returns

SUCCEED or FAIL.

SUCCEED 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 **dbcursorfetch** to FAIL.

FAIL is returned if at least one of the following is true:

- A *fetchtype* of FETCH_RANDOM, FETCH_RELATIVE, or FETCH_LAST was used on a dynamic cursor.
- A fetchtype other than FETCH_FIRST or FETCH_NEXT 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 **dbcursoropen**.

After the fetch, the elements of the array of row status indicators (*pstatus* in **dbcursoropen**) 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 joined in a logical OR. The following table shows the meaning of each row status value.

Fetch status	Description

FTC_SUCCEED	Row was successfully fetched. The array of bound program variables and the array of data length values (specified in dbcursorbind) contain valid data for the row.
FTC_MISSING	Row has been deleted or a unique index column of the row has been changed. Do not use the values in the array of bound program variables and the array of data length values (specified in dbcursorbind) 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.
FTC_ENDOFKEYSET	Is the end of the keyset. This fetch status is set for backward compatibility with "mixed" client cursors used by existing applications.
FTC_ENDOFRESULTS	Is the end of the result 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 the values in the array of bound program variables and the array of data length values (specified in dbcursorbind) for those rows.
	If joined in a logical OR with FTC_SUCCEED, this is the last row in the cursor result set; it contains valid data.
	If joined in a logical OR with FTC_MISSING, this is the last row in the cursor result set, but the row is missing.
	If not joined with FTC_SUCCEED or FTC_MISSING, this row is invalid.

A row status indicator of 0 means that the row is invalid, and the values in the

array of bound program variables and the array of data length values (specified in **dbcursorbind**) do not contain valid data. This usually happens when the row is before the beginning (first row) or after the end (last row) of the cursor result set.

After the fetch, the elements of the array of bound program variables and the array of data length values (specified earlier in **dbcursorbind**):

- Are filled with valid data for all rows with a fetch status of FTC_SUCCEED.
- Contain invalid data for all rows without a fetch status of FTC_SUCCEED.

If no fetches have been performed on a cursor, the current cursor position is before the beginning (first row) of the cursor result 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 result set.
- Adjusted to the first row of the cursor result set if the first row of the new fetch buffer would have been before the first row of the cursor result set and the last row of the new fetch buffer would have stayed within the cursor result set due to a FETCH_PREV operation.
- Unchanged from the current position if all rows of the new fetch buffer would have been before the first row of the cursor result set, if the first row (and thus all rows) of the new fetch buffer would have been after the last row of the cursor result 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 new fetch buffer would have been before the first row of the set and the last row of the new fetch buffer would have stayed within the cursor result set due to a FETCH_RELATIVE 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

result set, all rows in the fetch buffer are invalid and will not have a fetch status of FTC_SUCCEED. In the case of dynamic cursors, the first row of the fetch buffer will have a fetch status of FTC_ENDOFRESULTS, 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 result set, all rows in the fetch buffer are invalid and will not have a fetch status of FTC_SUCCEED. The first row of the fetch buffer will have a fetch status of FTC_ENDOFRESULTS, 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 **dbcursor** 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 **dbcursorfetch** leaves the DBPROCESS structure available for use with no pending results.

CAUTION This function works with client cursors and transparent server cursors. Do not use both **dbcursorfetch** and **dbcursorfetchex** with the same cursor handle. After one of these functions is used on a specific cursor handle, any attempt to use the other function returns FAIL.

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 result set (which happens when the first row of the new fetch buffer would have been before the first row of the dynamic cursor result set and the last row of the new fetch buffer would have stayed within the dynamic cursor result set due to a FETCH_PREV 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

result set or includes rows from the current fetch buffer again.

Transparent server cursor

A *fetchtype* of FETCH_NEXT or FETCH_PREV using a dynamic cursor is actually mapped to a relative fetch on SQL Server 6.0 or later. Because of this, if the first row in the current fetch buffer is deleted before a FETCH_NEXT (mapped to a forward relative fetch on SQL Server 6.0 or later) is performed, the current cursor position becomes invalid. For more information about the fetch behavior in this case, see **dbcursorfetchex**.

See Also

Bulk-Copy Functions dbcursorcolinfo dbcursor dbcursorinfo dbcursorbind dbcursoropen dbcursorclose

dbcursorfetchex

Fetches a block of rows (called the fetch buffer) from an explicit server cursor and stores the rows in the bound program variables established using **dbcursorbind**.

Syntax

RETCODE dbcursorfetchex (PDBCURSOR hc, INT fetchtype, DBINT rownum, DBINT nfetchrows, DBINT reserved)

Arguments

hc

Is a PDBCURSOR pointer to a server cursor returned by **dbcursoropen**.

fetchtype

Specifies the type of fetch to execute, changing the position of the fetch buffer within the cursor result set. The following table describes the different *fetchtype* values.

fetchtype	Description
FETCH_FIRST	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 result set.
FETCH_NEXT	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 FETCH_FIRST.

FETCH_PREV	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 dbcursoropen) before the first row of the current fetch buffer.
FETCH_RANDOM	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 result set.
FETCH_RELATIVE	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.
FETCH_LAST	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 result 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 CUR_FORWARD in **dbcursoropen**), you can only use the FETCH_FIRST, FETCH_NEXT, or FETCH_RELATIVE (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 FETCH_RANDOM or FETCH_RELATIVE. Specify 0 for any other *fetchtype*.

When *fetchtype* is FETCH_RANDOM:

- 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 result set.
- A negative *rownum* means that the first row of the new fetch buffer is *rownum* rows backward from the end of the cursor result set. Given *n* rows in the cursor result set, the first row of the new fetch buffer is row
n+1+*rownum* of the cursor result 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 result 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 result set.

• A *rownum* of 0 means that the first row of the new fetch buffer is before the beginning (first row) of the cursor result set.

When *fetchtype* is FETCH_RELATIVE:

• 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 nondeleted 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 earlier, 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 nondeleted 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 Microsoft® SQL Server[™] without moving the current cursor position. This is identical to calling **dbcursor** with *optype* set to CRS_REFRESH.

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 **dbcursoropen**. 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 **dbcursorfetchex**.

When *fetchtype* is FETCH_FIRST, an *nfetchrows* value of 0 means that the new cursor position is set to before the beginning (first row) of the cursor result set.

When *fetchtype* is FETCH_LAST, an *nfetchrows* value of 0 means that the new cursor position is set to after the end (last row) of the cursor result set.

reserved

Reserved for future use. Use 0.

Returns

SUCCEED or FAIL.

SUCCEED 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 **dbcursorfetchex** to FAIL.

FAIL is returned if at least one of the following is true:

- A *fetchtype* of FETCH_RANDOM was used on a dynamic cursor.
- A fetchtype other than FETCH_FIRST, FETCH_NEXT, or FETCH_RELATIVE (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 **dbcursoropen**) 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 joined in a logical OR. The following table shows the meaning of each row status value.

Fetch status	Description
FTC_SUCCEED	Row was successfully fetched. The array of bound program variables and the array of data length values (specified in dbcursorbind) contain valid data for the row.
FTC_MISSING	Row has been deleted or a unique index column of the row has been changed. Do not use the values in the array of bound program variables and the array of data length values (specified in dbcursorbind) 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 the values in the array of bound program variables and the array of data length values (specified in **dbcursorbind**) do not contain valid data. This happens when the row is before the beginning (first row) or after the end (last row) of the cursor result set.

After the fetch, the elements of the array of bound program variables and the array of data length values (earlier specified in **dbcursorbind**):

• Are filled with valid data for all rows with a fetch status of FTC_SUCCEED.

• Contain invalid data for all rows *without* a fetch status of FTC_SUCCEED.

If no fetches have been performed on a cursor, the current cursor position is before the beginning (first row) of the cursor result 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 result set.
- Adjusted to the first row of the cursor result set if the first row of the new fetch buffer would have been before the first row of the cursor result set and the last row of the new fetch buffer would have stayed within the cursor result set.
- Before the beginning of the cursor result set if all rows of the new fetch buffer are before the first row of the cursor result set, or if any backward fetch (FETCH_PREV or FETCH_RELATIVE with a negative *rownum*) is performed when the first row of the current fetch buffer is the first row of the cursor result set.
- After the end of the cursor result set if the first row (and thus all rows) of the new fetch buffer is after the last row of the cursor result set.

When the current cursor position is before the beginning of the cursor, a FETCH_NEXT operation is identical to a FETCH_FIRST operation. When the current cursor position is after the end of the cursor, a FETCH_PREV operation is identical to a FETCH_LAST operation.

Note This function works with explicit server cursors in SQL Server version 6.0 or later. Do not use both **dbcursorfetchex** and **dbcursorfetch** with the same server cursor handle. After one of these functions is used on a specific cursor handle, any attempt to use the other function returns FAIL.

Each call to **dbcursorfetch** leaves the DBPROCESS structure available for use with no pending results.

See Also

Bulk-Copy Functions

<u>dbcursorcolinfo</u>

<u>dbcursor</u>

<u>dbcursorinfo</u>

<u>dbcursorbind</u>

<u>dbcursoropen</u>

<u>dbcursorclose</u>

dbcursorinfo

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 **dbcursorinfoex** function returns more detailed information.

Syntax

RETCODE dbcursorinfo (PDBCURSOR hcursor, **LPINT** ncols, **LPDBINT** nrows);

Arguments

hcursor

Is the cursor handle created by **dbcursoropen**.

ncols

Is a pointer to a program variable that will contain the number of columns in the cursor.

nrows

Is a pointer to a program variable that will contain the number of rows in a cursor result 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 result 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 result set is incomplete, or the total number of rows in the cursor result set. You can call **dbcursorinfoex** to determine this.

Returns

SUCCEED or FAIL.

Remarks

Call **dbcursorinfoex** for more complete information about transparent server cursors and explicit server cursors.

See Also

Bulk-Copy Functions dbcursorcolinfo dbcursor dbcursorfetch dbcursorbind dbcursoropen dbcursorclose

dbcursorinfoex

Returns information about a client cursor, a transparent server cursor, or an explicit server cursor.

Syntax

RETCODE dbcursorinfoex(PDBCURSOR *hc*, **LPDBCURSORINFO** *pdbcursorinfo*)

Arguments

hc

Is a PDBCURSOR pointer to a server cursor returned by **dbcursoropen**.

pdbcursorinfo

Is a pointer to a DBCURSORINFO structure that DB-Library will fill with information about the specified cursor.

The DBCURSORINFO structure is defined as follows:

```
typedef struct
{
    DBINT SizeOfStruct;
    ULONG TotCols;
    ULONG TotRows;
    ULONG CurRow;
    ULONG CurRow;
    ULONG TotRowsFetched;
    ULONG Type;
    ULONG Status;
} DBCURSORINFO, PTR LPDBCURSORINFO;
```

The DBCURSORINFO fields are described in the table.

Field	Description
SizeOfStruct	Before calling dbcursorinfoex , set this field equal to the value returned by the C sizeof function for the DBCURSORINFO structure.
TotCols	Is the total number of columns in the cursor.
TotRows	Is the total number of rows in the cursor result 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 result set; otherwise, - 1 is returned. Transparent server cursor, explicit server cursor:
	For a dynamic cursor, - 1 is returned. For a keyset cursor, if the Status field is CU_FILLING, the asynchronous population of the cursor result set is incomplete, and this value indicates the number of rows populated. If the Status field is CU_FILLED, the cursor result set is completely populated, and this value indicates the total number of rows in the cursor result set.
CurRow	Is the row number within the cursor result set of the current cursor position (first row of the fetch buffer). The first row of the cursor result set is number 1.
	Transparent server cursor, explicit server cursor: For a keyset cursor, this value is always valid. For a dynamic cursor, this value will always be 1 when the current position is within the cursor result set. 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.
TotRowsFetched	Is the total number of valid rows in the current fetch

	buffer.
Туре	Is a bitmap of cursor type, scroll option, and concurrency control information. It is a series of the following values joined in a logical OR: Type: CU_CLIENT: Client cursor CU_SERVER: Transparent server cursor or explicit server cursor
	Scroll option: CU_DYNAMIC: Dynamic cursor CU_FORWARD: Forward-only, dynamic cursor CU_KEYSET: Keyset cursor CU_INSENSITIVE: Insensitive keyset cursor CU_MIXED: Mixed-mode cursor (provided for backward compatibility only)
	Concurrency control: CU_READONLY: Read-only concurrency CU_LOCKCC: Intent to update concurrency CU_OPTCC: Optimistic concurrency based on timestamp or values CU_OPTCCVAL: Optimistic concurrency based on values
Status	Bitmap of status information. It is a series of the following values joined in a logical OR: Client cursor: CU_FILLED: All cursors
	Transparent server cursor, explicit server cursor: CU_FILLING: Incomplete asynchronous population of a keyset transparent server cursor or keyset explicit server cursor result set CU_FILLED: Incomplete asynchronous population of a keyset transparent server cursor or keyset explicit server cursor result set, or the cursor is a dynamic cursor

Returns

SUCCEED or FAIL.

Remarks

Before calling **dbcursorinfoex**, set the **SizeOfStruct** field equal to the value returned by the C **sizeof** function for the DBCURSORINFO structure. The **dbcursorinfoex** function fills the supplied DBCURSORINFO structure with information about the open server cursor.

See Also

Bulk-Copy Functions dbcursorclose dbcursor dbcursorcolinfo dbcursorbind dbcursorfetch

dbcursoropen

Opens a cursor. It is an explicit server cursor if you are connected to Microsoft® SQL Server[™] version 6.0 or later and **dbcursorfetchex** is used for the first fetch. It is a transparent server cursor if you are connected to SQL Server 6.0 or later and **dbcursorfetch** 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 **dbcursorfetch** can be used to fetch rows.

Syntax

PDBCURSOR dbcursoropen (PDBPROCESS dbproc, LPCSTR stmt, INT scrollopt, INT concuropt, UINT nrows, LPDBINT pstatus);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

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 CUR_KEYSET, 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 CUR_KEYSET cursor will be used if a *scrollopt* of CUR_FORWARD or CUR_DYNAMIC is requested. This also occurs if the SELECT contains a subquery.

The cursor is automatically opened with a *scrollopt* of CUR_INSENSITIVE and a *concuropt* of CUR_READONLY 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.

scrollopt	Description
CUR_DYNAMIC	Dynamic cursor.
	Client cursor, transparent server cursor: The dbcursorfetch function will allow only a <i>fetchtype</i> of FETCH_FIRST, FETCH_NEXT, or FETCH_PREV.
	Explicit server cursor: The dbcursorfetchex function will allow all <i>fetchtype</i> values except FETCH_RANDOM.
CUR_FORWARD	Forward-only, dynamic cursor.
	Client cursor, transparent server cursor: The dbcursorfetch function will allow only a <i>fetchtype</i> of FETCH_FIRST or FETCH_NEXT. Explicit server cursor: The dbcursorfetchex function will allow only a <i>fetchtype</i> of FETCH_FIRST, FETCH_NEXT, or FETCH_RELATIVE with a positive <i>rownum</i> .
CUR_KEYSET	Keyset cursor. The dbcursorfetch and dbcursorfetchex functions will allow all <i>fetchtype</i> values.
CUR_INSENSITIVE	 Client cursor: Not supported. Transparent server cursor, explicit server cursor: Insensitive keyset cursor. Use a <i>concuropt</i> of CUR_READONLY. SQL Server will generate a

	others will not be visible through the cursor.
	The dbcursorfetch and dbcursorfetchex functions will allow all <i>fetchtype</i> values.
n > 1	Client cursor: For backward compatibility with "mixed" client cursors.
	Transparent server cursor, explicit server cursor: Mapped to a CUR_KEYSET cursor.

concuropt

Is one of the following concurrency control options.

concuropt	Description
CUR_READONLY	Read-only cursor. You cannot modify rows in the
	cursor result set.
CUR_LOCKCC	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.
CUR_OPTCC	Optimistic concurrency control using timestamp or values. Changes to a row that are initiated 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.
CUR_OPTCCVAL	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 **dbcursorfetch**.

Explicit server cursor:

Is the maximum number of rows in the fetch buffer. The *nfetchrows* parameter of **dbcursorfetchex** must be less than or equal to this value.

pstatus

Is a pointer to the array of row status indicators. This array must contain *nrows* DBINT elements. A row status value is a bitmap of fetch status values joined in a logical OR.

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 **dbcursorfetch**.

Explicit server cursor:

For more information about fetch status values, see **dbcursorfetchex**.

Returns

A handle to the cursor if the cursor open succeeds. If it fails, NULL 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 result 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 **dbcursoropen** returns a valid cursor handle, you can call **dbcursorinfoex** 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 result set is determined at fetch time, and it can change between each fetch. A row disappears from the cursor result 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 result 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 result set is fixed at open time. A row is marked as missing from the cursor result 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 result set only if it 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 DBPROCESS connection. When cursor functions are called, there should be no commands waiting to be executed or results pending in the DBPROCESS connection.

When you call the **dbcursoropen** function with a *scrollopt* parameter of CUR_DYNAMIC, the tables you specify in the SELECT statement of the *stmt* parameter no longer require unique indexes.

In SQL Server 6.0, the cursor automatically opens with a *scrollopt* parameter of CUR_INSENSITIVE and a *concuropt* parameter of CUR_READONLY if the SELECT statement contains a table without a unique index.

In SQL Server 6.5 or later, the cursor automatically opens with a *scrollopt* parameter of CUR_INSENSITIVE and a *concuropt* parameter of CUR_READONLY if the SELECT statement contains any of the following:

- ORDER BY and a table with no unique index
- UNION
- DISTINCT
- GROUP BY
- HAVING
- Aggregate function

SQL Server copies the results into temporary work tables. Because all fetch operations are performed on the temporary tables, you cannot make changes to the cursor by using **dbcursor**.

See Also

dbcursordbcursorfetchdbcursorbinddbcursorinfodbcursorinfodbcursorclose

Bulk-Copy Functions

<u>dbcursorcolinfo</u>

Stored Procedure Functions

DB-Library for C version 7.0 supports several stored procedure functions.

dbhasretstat

Determines whether a stored procedure or a remote stored procedure generated a return status number.

Syntax

DBBOOL dbhasretstat (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

TRUE or FALSE.

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 **dbhasretstat** after **dbresults** returns NO_MORE_RPC_RESULTS (for all stored procedures in a batch except the last one) or NO_MORE_RESULTS (for a single stored procedure, or for the last stored procedure in a batch).

The **dbretstatus** function actually retrieves the status number. Stored procedures that complete generally 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

<u>dbnextrow</u>

<u>dbrpcinit</u>

<u>dbresults</u>

<u>dbrpcparam</u>

<u>dbretdata</u>

<u>dbrpcsend</u>

<u>dbretstatus</u>

dbnumrets

Calculates the number of returned parameter values generated by a stored procedure or a remote stored procedure.

Syntax

INT dbnumrets (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The number of parameter values associated with the most recently executed stored procedure.

Remarks

For more information about stored procedure return parameters, see **dbretdata**.

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 **dbnumrets** after **dbresults** returns NO_MORE_RPC_RESULTS (for all stored procedures in a batch except the last one) or NO_MORE_RESULTS (for a single stored procedure, or for the last stored procedure in a batch).

See Also

<u>dbnextrow</u>

<u>dbrettype</u>

<u>dbresults</u>

<u>dbrpcinit</u>

<u>dbretlen</u>

<u>dbrpcparam</u>

<u>dbretname</u>

dbretdata

Returns a pointer to a returned parameter value generated by a stored procedure or a remote stored procedure.

Syntax

LPCBYTE dbretdata (PDBPROCESS dbproc, INT retnum);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

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*, nonreturn 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 pointer to the specified return value. If *retnum* is out of range, **dbretdata** returns NULL. 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 **dbretlen**.

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 **dbretdata** after **dbresults** returns NO_MORE_RPC_RESULTS (for all stored procedures in a batch except the last one) or NO_MORE_RESULTS (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 within 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 **dbrpcinit**), when the parameter is added using **dbrpcparam**, the *status* bitmask must contain the DBRPCRETURN option.

If a stored procedure is executed using DB-Library functions (such as **dbrpcinit**), the return parameter values are automatically available to the application. If the 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

dbnextrow dbrettype dbresults <u>dbrpcinit</u>

<u>dbretname</u>

<u>dbrpcparam</u>

dbretlen

Determines the length of a return-parameter value generated by a stored procedure or a remote stored procedure.

Syntax

DBINT dbretlen (PDBPROCESS *dbproc*, **INT** *retnum* **)**;

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

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*, nonreturn 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, **dbretlen** returns -1. If the return value is null, **dbretlen** returns 0.

Remarks

For more information about stored procedure return parameters, see <u>dbretdata</u>.
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 **dbretlen** after **dbresults** returns NO_MORE_RPC_RESULTS (for all stored procedures in a batch except the last one) or NO_MORE_RESULTS (for a single stored procedure, or for the last stored procedure in a batch).

See Also

<u>dbnextrow</u>

<u>dbrettype</u>

<u>dbnumrets</u>

<u>dbrpcinit</u>

<u>dbresults</u>

<u>dbrpcparam</u>

<u>dbretdata</u>

dbretname

Determines the name of a return parameter of a stored procedure or a remote stored procedure.

Syntax

LPCSTR dbretname (PDBPROCESS *dbproc*, INT *retnum*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

retnum

Is the number of the return 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*, nonreturn 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 pointer to the null-terminated parameter name for the specified returnparameter value. If *retnum* is out of range, **dbretname** returns NULL.

Remarks

For more information about stored procedure return parameters, see <u>dbretdata</u>.

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 **dbretname** after **dbresults** returns NO_MORE_RPC_RESULTS (for all stored procedures in a batch except the last one) or NO_MORE_RESULTS (for a single stored procedure, or for the last stored procedure in a batch).

See Also

<u>dbnextrow</u>

<u>dbrettype</u>

<u>dbnumrets</u>

<u>dbrpcinit</u>

<u>dbresults</u>

<u>dbrpcparam</u>

<u>dbretdata</u>

dbretstatus

Determines the stored procedure status number returned by a stored procedure or a remote stored procedure.

Syntax

DBINT dbretstatus (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

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	Data type 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 is corrupt.
-13	Database is corrupt.
-14	Hardware error.

The values -15 through -99 are reserved for future SQL Server expansion.

The **dbhasretstat** function determines whether the most recently executed stored procedure actually 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 **dbretstatus** after **dbresults** returns NO_MORE_RPC_RESULTS (for all stored procedures in a batch except the last one) or NO_MORE_RESULTS (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 returnparameter values is unimportant.

See Also

dbhasretstat dbrpcinit dbnextrow dbrpcparam dbresults dbrpcsend

dbretdata

dbrettype

Determines the data type of a return-parameter value generated by a stored procedure or a remote stored procedure.

Syntax

INT dbrettype (PDBPROCESS dbproc, INT retnum);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

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*, nonreturn 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 data type of the specified return value.

Column data type	Returned constant	
Char	SQLCHAR	
Varchar	SQLCHAR	
Binary	SQLBINARY	

Varbinary	SQLBINARY
Tinyint	SQLINT1
Smallint	SQLINT2
Int	SQLINT4
Real	SQLFLT4
Float	SQLFLT8
smallmoney	SQLMONEY4
Money	SQLMONEY
Decimal	SQLDECIMAL
Numeric	SQLNUMERIC
smalldatetime	SQLDATETIM4
Datetime	SQLDATETIME

If *retnum* is out of range, -1 is returned.

Remarks

For more information about stored procedure return parameters, see <u>dbretdata</u>.

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 **dbrettype** after **dbresults** returns NO_MORE_RPC_RESULTS (for all stored procedures in a batch except the last one) or NO_MORE_RESULTS (for a single stored procedure, or for the last stored procedure in a batch).

The **dbrettype** function actually returns an integer token value for the data type (SQLCHAR, SQLFLT8, and so on). To convert the token value into a readable token string, use **dbprtype**. For more information about a list of all token values and their equivalent token strings, see <u>dbprtype</u>.

See Also

<u>dbnextrow</u>

<u>dbretlen</u>

<u>dbnumrets</u>

<u>dbretname</u>

<u>dbprtype</u>

<u>dbrpcinit</u>

<u>dbresults</u>

<u>dbrpcparam</u>

<u>dbretdata</u>

DB-Library for C Data Types

dbrpcexec

Executes a single stored procedure, a single remote stored procedure, or batch of stored procedures or remote stored procedures on Microsoft® SQL ServerTM 2000.

Syntax

RETCODE dbrpcexec (PDBPROCESS *pdbproc*);

Arguments

pdbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL.

Remarks

After initializing and setting up each stored procedure using **dbrpcinit** and **dbrpcparam**, call **dbrpcexec** to execute the stored procedure or batch of stored procedures on SQL Server. Then call **dbsqlok** before processing the stored procedure results. **dbrpcexec** can be faster than **dbrpcsend** on some networks. For more information about executing stored procedures using DB-Library functions, see **dbrpcinit**.

Note This function is one of the five (**dbnextrow**, **dbresults**, **dbrpcexec**, **dbsqlexec**, and **dbsqlok**) 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 **dbrpcsend** and **dbsqlok**, or set the DB-Library time-out to regain control periodically.

See Also

<u>dbnextrow</u>

<u>dbretstatus</u>

<u>dbresults</u>

<u>dbrpcparam</u>

<u>dbretdata</u>

<u>dbsqlok</u>

dbrpcinit

Initializes a stored procedure or a remote stored procedure.

Syntax

RETCODE dbrpcinit (PDBPROCESS dbproc, LPCSTR rpcname, DBSMALLINT options);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

rpcname

Is a pointer to 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
DBRPCRECOMPILE	Recompiles a stored procedure before it is executed.
DBRPCRESET	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 cancellation 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 **dbrpcinit**) 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 **dbrpcinit** once to initialize a new stored procedure.
- 2. Call **dbrpcparam** 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 **dbrpcsend** or **dbrpcexec** to send the entire stored procedure batch to SQL Server.
- 5. Call **dbsqlok** to wait for SQL Server to start returning results.
- 6. Call **dbresults** to process the results from each stored procedure.

If **dbresults** returns SUCCEED, call **dbnextrow** until it returns NO_MORE_ROWS to process the normal results from the stored procedure.

If **dbresults** returns NO_MORE_RPC_RESULTS, and you want to retrieve status number and return-parameter information returned by the stored procedure, follow the steps given later.

7. Repeat Step 6 until **dbresults** returns NO_MORE_RESULTS.

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 later.

After **dbresults** returns NO_MORE_RPC_RESULTS (for all stored procedures in a batch except the last one) or NO_MORE_RESULTS (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 **dbretstatus** and **dbhasretstat** to retrieve the return status number.
- Call **dbnumrets** to determine the number of return parameters.
- Call **dbretdata**, **dbrettype**, **dbretlen**, and **dbretname** 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 passes parameters in their native data types; using an EXECUTE statement passes parameters as ASCII characters. Calling stored procedures with DB-Library functions works faster and usually more efficiently than an EXECUTE statement, because neither the application nor the server is required to convert native data types 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 **dbretdata**. (Note, however, that a return parameter must be specified as such when it is first added to the stored procedure through **dbrpcparam**.)

If 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 *dbproc* connection) generally participate in transactions and can be rolled back. Remote stored procedures executed on a remote SQL Server cannot be rolled back.

In SQL Server version 6.5 or later, you can pass stored procedure parameters when you open a cursor by calling the **dbrpcinit** function and the **dbrpcparam** function.

The **dbrpcinit** function has an *option* parameter: **DBRPCCURSOR**. Use **DBRPCCURSOR** to specify the initialization of an input-parameter list for a subsequent cursor-open operation on a stored procedure. When you use **DBRPCCURSOR**, you must set the *rpcname* parameter to NULL.

No pending remote procedure calls (RPCs) can exist when you call **dbrpcinit** with an *option* of **DBRPCCURSOR**. The **dbrpcinit** function returns FAIL if pending RPCs have been initiated.

You can call the **dbrpcparam** function to set the value for each stored procedure parameter that was used in the process of opening a cursor. Stored procedure parameters are no longer required to be constants, and they can be passed as follows.

To open a cursor on a stored procedure that passes program variables as parameters

1. Call the **dbrpcinit** function once to initialize a new stored procedure. Set the *rpcname* parameter to NULL, and set the *option* parameter to **DBRPCCURSOR**.

- 2. Call the **dbrpcparam** function once per parameter to set the value of each parameter of the stored procedure.
- 3. Call the **dbcursoropen** function to open the cursor, and in the *stmt* parameter, specify only the stored procedure name.
- 4. Call the remaining cursor functions as you ordinarily would.

See Also

<u>dbnextrow</u>

<u>dbrpcparam</u>

<u>dbresults</u>

<u>dbrpcsend</u>

<u>dbretdata</u>

<u>dbsqlok</u>

<u>dbretstatus</u>

dbrpcparam

Adds a parameter to a stored procedure or a remote stored procedure.

Syntax

RETCODE dbrpcparam (PDBPROCESS dbproc, LPCSTR paramname, BYTE status, INT type, DBINT maxlen, DBINT datalen, LPCBYTE value);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

paramname

Is a pointer to the name of the parameter to be invoked. This name must begin with (@), as do all parameter names within stored procedures. As in the Transact-SQL EXECUTE statement, the name is optional. If no name is used, this parameter should be specified as NULL. In that case, the order of the **dbrpcparam** 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 DBRPCRETURN, which signifies that an application designates this parameter as a return parameter. Specify 0 to indicate no options.

type

Is the data type 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), *maxlen* is 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 (where *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 non-NULL 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 pointer to the program variable containing the stored procedure parameter value itself.

The following table summarizes the required *maxlen* and *datalen* values for each type of parameter.

Parameter	maxlen	datalen
Fixed-length	- 1	- 1
Variable-length	Maximum desired	Length of input value not
	length of return value.	counting null terminator.
Fixed-length NULL	- 1	0
Variable-length NULL	0	0

When specifying a NULL parameter, the actual contents of *value* are not used, and a NULL parameter is added to the stored procedure. However, when *type* is SQLDECIMAL or SQLNUMERIC, *value* must still point to a valid DBDECIMAL or DBNUMERIC structure.

Returns

SUCCEED or FAIL.

Remarks

After initializing a stored procedure using **dbrpcinit**, you must call **dbrpcparam** 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 <u>dbrpcinit</u>.

See Also

<u>dbnextrow</u>

<u>dbretstatus</u>

<u>dbresults</u>

<u>dbrpcinit</u>

<u>dbretdata</u>

<u>dbsqlok</u>

dbrpcsend

Sends a single stored procedure, a single remote stored procedure, or batch of stored procedures or remote stored procedures to Microsoft® SQL ServerTM 2000 to be executed.

Syntax

RETCODE dbrpcsend (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL.

Remarks

After initializing and setting up each stored procedure using **dbrpcinit** and **dbrpcparam**, call **dbrpcsend** to send the stored procedure or batch of stored procedures to SQL Server. Then call **dbsqlok** before processing the stored procedure results. For more information about executing stored procedures using DB-Library functions, see <u>dbrpcinit</u>.

See Also

dbnextrow dbretstatus dbresults <u>dbrpcinit</u>

<u>dbretdata</u>

<u>dbsqlok</u>

Text and Image Functions

DB-Library for C version 7.0 supports several **text** and **image** functions.

dbmoretext

Sends part of a **text** or **image** value to Microsoft® SQL Server[™] 2000.

Syntax

RETCODE dbmoretext (PDBPROCESS *dbproc*, **DBINT** *size*, **LPCBYTE** *text* **)**;

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

size

Is the size, in bytes, of the particular part of the **text** or **image** value being sent to SQL Server. It is an error to send more **text** or **image** bytes to SQL Server than were specified in the call to **dbwritetext** or **dbupdatetext**.

text

Is a pointer to the **text** or **image** portion to be written.

Returns

SUCCEED or FAIL.

Remarks

This function is used in conjunction with **dbwritetext** or **dbupdatetext** to send a large SQLTEXT or SQLIMAGE value to SQL Server in the form of a number of smaller chunks. This is particularly useful with operating systems that cannot

allocate extremely long data buffers. A **text** or **image** fragment must be shorter than 64 KB.

After calling **dbmoretext** for the last time, call **dbsqlok**.

For an example of **dbmoretext**, see <u>dbwritetext</u>.

See Also

<u>dbtxptr</u>

<u>dbwritetext</u>

dbtxtimestamp

dbreadtext

Reads part of a **text** or **image** value from Microsoft® SQL Server[™] 2000.

Syntax

DBINT dbreadtext (PDBPROCESS *dbproc*, **LPVOID** *buf*, **DBINT** *bufsize* **)**;

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

buf

Is a pointer to a caller-allocated buffer that will contain the chunk of **text** or **image** data.

bufsize

Is the size, in bytes, of the caller's buffer. This value must be smaller than 64 KB.

Returns

Returns	Description
>0	The number of bytes put into the caller's buffer.
0	The end of a row.
-1	An error.
NO_MORE_ROWS	All rows read.

Remarks

Use **dbreadtext** instead of **dbnextrow** to read SQLTEXT and SQLIMAGE values.

This function takes a large SQLTEXT or SQLIMAGE value from SQL Server and breaks it into several smaller chunks. A buffer cannot be larger than 64 KB although the **text** or **image** value can be much larger.

To read successive chunks of the same SQLTEXT or SQLIMAGE value, call **dbreadtext** until it returns 0 (end of row).

The **dbreadtext** function processes the results of Transact-SQL queries if those queries return only one column containing either **text** or **image** data. The Transact-SQL READTEXT statement returns results of this type.

See Also

Bulk-Copy Functions
dbwritetext
dbmextrow
dbtxptr

Returns the value of the **text** or **image** pointer for a column in the current row.

Syntax

LPCDBBINARY dbtxptr (PDBPROCESS dbproc, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the column. The first column in a table is number 1. For further information, see <u>dbadata</u>.

Returns

A DBBINARY pointer to the **text** or **image** pointer for the column. This pointer can be NULL.

Remarks

Every database column of type SQLTEXT or SQLIMAGE has an associated text pointer that uniquely identifies the **text** or **image** value. Use this text pointer with **dbwritetext**. Call **dbtxptr** only after **dbnextrow** or **dbgetrow** has returned REG_ROW.

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

Text pointers are of fixed length and can be NULL when the **text** or **image** value is NULL.

See Also

<u>dbwritetext</u>

dbtxtimestamp

Returns the value of the **text** timestamp for a column in the current row.

Syntax

LPCDBBINARY dbtxtimestamp (PDBPROCESS dbproc, INT column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

column

Is the number of the column. The first column in a table is number 1. For further information, see <u>dbadata</u>.

Returns

A DBBINARY pointer to the **text** timestamp for the column. This pointer can be NULL when the **text** or **image** value is null, or when there is no current row.

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

Remarks

Every database column of type SQLTEXT or SQLIMAGE has an associated **text** timestamp, which marks the time of the column's last modification. Use the **text** timestamp with **dbwritetext** to ensure that one user doesn't inadvertently overwrite another's modifications to the same value in the database. The **text** timestamp is returned to the DBPROCESS when a Transact-SQL SELECT

statement is performed on a SQLTEXT or SQLIMAGE column.

The length of a non-NULL **text** timestamp is always DBTXTSLEN (currently defined as eight bytes). Call **dbtxtimestamp** only after **dbnextrow** or **dbgetrow** has returned REG_ROW.

See Also

<u>dbreadtext</u>

<u>dbwritetext</u>

dbtxtsnewval

Returns the new value of a **text** timestamp after a call to **dbwritetext**.

Syntax

LPCDBBINARY dbtxtsnewval (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

A pointer to the new **text** timestamp value for the SQLTEXT or SQLIMAGE value modified by a **dbwritetext** operation. This pointer can be NULL.

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

Remarks

Every database column of type SQLTEXT or SQLIMAGE has an associated **text** timestamp, which is updated whenever the column's value is changed. Use the **text** timestamp with **dbwritetext** to ensure that one user doesn't inadvertently overwrite another's modifications to the same value in the database. It is returned to the DBPROCESS when a Transact-SQL SELECT statement is performed on a SQLTEXT or SQLIMAGE column and can be examined by calling **dbtxtimestamp**.

After each successful **dbwritetext** operation (which can include a number of calls to **dbmoretext**), SQL Server sends the updated **text** timestamp value back to DB-Library. The **dbtxtsnewval** function lets the application get this new

timestamp value. The application can then use **dbtxtsput** to put the new **timestamp** value in the DBPROCESS for future access through **dbtxtimestamp**.

The application can use **dbtxtsnewval** in two ways. First, the return from **dbtxtsnewval** can be used as the *timestamp* parameter of a **dbwritetext** call. Second, **dbtxtsnewval** and **dbtxtsput** can be used together to put the new **timestamp** value into the DBPROCESS row buffer for future access using **dbtxtimestamp**. This is particularly useful when the application is buffering result rows and does not need the new timestamp immediately.

See Also

<u>dbmoretext</u> <u>dbwritetext</u>

<u>dbreadtext</u>

dbtxtsput

Puts the new value of a **text** timestamp into the specified column of the current row in the DBPROCESS.

Syntax

RETCODE dbtxtsput (PDBPROCESS dbproc, LPCDBBINARY newtxts, INT colnum);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

newtxts

Is a pointer to the new **text** timestamp value. It is returned by **dbtxtsnewval**.

colnum

Is the number of the column associated with this **text** timestamp. Column numbers start at 1.

Returns

SUCCEED or FAIL.

Remarks

Every database column of type SQLTEXT or SQLIMAGE has an associated **text** timestamp, which is updated whenever the column's value is changed. Use the **text** timestamp with **dbwritetext** to ensure that one user doesn't inadvertently

overwrite another's modifications to the same value in the database. It is returned to the DBPROCESS when a Transact-SQL SELECT statement is performed on a SQLTEXT or SQLIMAGE column and can be examined by calling **dbtxtimestamp**. Call **dbtxtsput** only after **dbnextrow** or **dbgetrow** has returned REG_ROW.

After each successful **dbwritetext** operation (which can include a number of calls to **dbmoretext**), SQL Server sends the updated **text** timestamp value back to DB-Library. The **dbtxtsnewval** function allows the application to get this new **timestamp** value. The application can then use **dbtxtsput** to put the new **timestamp** value in the DBPROCESS row buffer for future access through **dbtxtimestamp**. This is particularly useful when the application is buffering result rows and does not need the new timestamp immediately.

See Also

dbmoretext dbtxtimestamp dbreadtext dbwritetext

dbupdatetext

Updates an existing **text** or **image** value. Unlike **dbwritetext**, which replaces an entire **text** or **image** value, **dbupdatetext** can change only a portion of a **text** or **image** value in place.

Syntax

RETCODE dbupdatetext (PDBPROCESS pdbproc, LPCSTR dest_object, LPCDBBINARY dest_textptr, LPCDBBINARY dest_timestamp, INT update_type, DBINT insert_offset, DBINT delete_length, LPCSTR src_object, DBINT src_size, LPCBYTE src_text);

Arguments

pdbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

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 **dbtxptr** to get this value. This parameter cannot be NULL.

dest_timestamp

Is the timestamp of the existing **text** or **image** value to be updated. Call **dbtxtimestamp** to get this value. This parameter cannot be NULL.

update_type

The type of update operation to perform.

Use UT_TEXT to insert new data from a C program variable specified in this call to **dbupdatetext**. You must specify an *src_object* value of NULL, an *src_size* equal to the size in bytes of the new data value being inserted, and a non-NULL *src_text* that points to the C program variable that contains the new data value being inserted.

Use UT_MORETEXT to insert new data from a C program variable with later calls to **dbmoretext**. You must specify a *src_object* value of NULL, a *src_size* equal to the total size in bytes of the new data value (that will be inserted by calls to **dbmoretext**), and a *src_text* value of NULL. Then call **dbmoretext** to insert the new data value in chunks.

Use UT_TEXTPTR to insert new data from a **text** or **image** column of an existing table. You must specify a non-NULL *src_object* value that gives the table and column, a *src_size* of 0, and a non-NULL *src_text* that gives the text pointer (returned by **dbtxptr**) of the new data value being inserted from the *src_object*.

Use UT_DELETEONLY to only delete existing data. You must specify an *insert_offset* value other than -1 and a nonzero *delete_length* value. Specify that no new data be inserted by using a *src_object* value of NULL, a *src_size* of 0 and a *src_text* value of NULL.

By default, a **dbupdatetext** operation is not recorded in the transaction log. You can join the UT_LOG bit flag in a bitwise OR with any of these *update_type* values (for example UT_TEXT | UT_LOG) 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 non-NULL, the *src_size* and *src_text* parameters are ignored. If NULL, 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 NULL.

src_text

Is a pointer to the source data value to be inserted. Use this parameter only if the *src_object* parameter is NULL.

Returns

SUCCEED or FAIL.

Remarks

The **dbupdatetext** 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 UT_DELETEONLY, an *insert_offset* value other than -1, a nonzero *delete_length* value, and the new data to be inserted.

To delete only existing data, specify an *update_type* value of UT_DELETEONLY, an *insert_offset* value other than -1 and a nonzero *delete_length* value. Do not specify any new data to be inserted.

To insert only new data, specify an *update_type* other than UT_DELETEONLY, a *delete_length* value of 0, and the new data to be inserted.

See Also

dbmoretext dbtxptr dbreadtext dbtxtimestamp dbresults dbtxtsnewval

<u>dbsqlok</u>

dbwritetext

Sends a **text** or **image** value to Microsoft[®] SQL Server[™] 2000.

Syntax

RETCODE dbwritetext (PDBPROCESS dbproc, LPCSTR objname, LPCDBBINARY textptr, DBTINYINT textptrlen, LPCDBBINARY timestamp, BOOL log, DBINT size, LPCBYTE text);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

objname

Is the database table name and column name. Separate the table name from the column name with a period.

textptr

Is the text pointer of the **text** or **image** value to be modified. Call **dbtxptr** to get this pointer. The pointer cannot be NULL.

textptrlen

Is a parameter included for future compatibility. For now, define its value with the constant DBTXPLEN.

timestamp

Is the **text** timestamp of the **text** or **image** value to be modified. Call **dbtxtimestamp** to get this timestamp. This value changes whenever the **text** or **image** value itself is changed. The pointer cannot be NULL.

log

Is a Boolean value, specifying whether this **dbwritetext** 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 pointer to the **text** or **image** to be written. If this pointer is NULL, DB-Library expects the application to call **dbmoretext** one or more times until all *size* bytes of data have been sent to SQL Server. For 16-bit DB-Library applications, no single data block can be 64 KB or larger. (Win32-based DB-Library applications are not limited to 64 KB data blocks.) DB-Library does not support huge pointers.

Returns

SUCCEED or FAIL.

Remarks

This function updates entire **text** and **image** values, allowing the application to send long values to SQL Server without having to copy them into a Transact-SQL UPDATE statement. In addition, it gives applications access to the **text** timestamp mechanism, which ensures that one user doesn't inadvertently overwrite another's modifications to the same value in the database.

The **dbwritetext** function succeeds only if its *timestamp* parameter, usually obtained when the column's value was originally retrieved, matches the **text** column's timestamp in the database. If a match occurs, **dbwritetext** updates the **text** column, and at the same time updates the column's timestamp. This has the effect of governing updates by competing application. An application's **dbwritetext** column between

the time the first application retrieved the column and the time it called **dbwritetext**.

The **dbwritetext** function is similar to the Transact-SQL WRITETEXT statement. It is usually more efficient to call **dbwritetext** than to send a WRITETEXT statement through the command buffer.

The **dbwritetext** function can be invoked with or without logging, according to the value of the *log* parameter. To use **dbwritetext** with logging turned off, the database option **select into/bulkcopy** must be set to true, as shown in the following example:

sp_dboption 'mbdb', 'select into/bulk copy ', 'true'

This function, in conjunction with the **dbmoretext** function, also allows the 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. All blocks must be shorter than 64 KB for 16-bit applications. (Win32-based DB-Library applications are not limited to 64 KB data blocks.) DB-Library does not support huge pointers.

When **dbwritetext** is used with **dbmoretext**, it locks the specified database **text** column, and the lock is not released until the final **dbmoretext** 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.

Examples

If the **text** parameter is a nonnull value, **dbwritetext** executes the data transfer from start to finish, including any necessary calls to **dbsqlok** and **dbresults**.

The following code fragment shows this use of **dbwritetext**:

LOGINREC *login; DBPROCESS *q_dbproc; DBPROCESS *u_dbproc; DBCHAR abstract_var[512];

// Open separate DBPROCESSes for querying and updating.

```
login = dblogin();
DBSETLUSER(login, "user");
DBSETLPWD(login, "my_passwd");
DBSETLAPP(login, "example1");
q_dbproc = dbopen(login, "my_server");
u dbproc = dbopen(login, "my_server");
```

// The database column "abstract" is a text column. Retrieve the
// value of one of its rows.

```
dbcmd(q_dbproc, "SELECT abstract FROM articles WHERE article_i
dbsqlexec(q_dbproc);
dbresults(q_dbproc);
dbbind(q_dbproc, 1, STRINGBIND, (DBINT)0, abstract_var);
```

```
while (dbnextrow(q_dbproc) != NO_MORE_ROWS)
{
    // Change the value of "abstract_var".
    strcpy(abstract_var, "A brand new value.");
    // Update the text column.
    dbwritetext (u_dbproc, "articles.abstract", dbtxptr(q_dbproc, 1),
        DBTXPLEN, dbtxtimestamp(q_dbproc, 1), TRUE,
        (DBINT)strlen(abstract_var), abstract_var);
}
// Done.
dbexit();
```

To send chunks of a **text** or **image** value, rather than the whole value at once, set the **text** parameter to NULL. Then, **dbwritetext** 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 **dbmoretext**, which can be called multiple times, once for each chunk. The following code fragment uses **dbwritetext** with **dbmoretext**:

```
LOGINREC *login;
DBPROCESS *q_dbproc;
DBPROCESS u_dbproc;
DBCHAR art1[512];
static DBCHAR part2[512] = "This adds another sentence to the text.
```

```
login = dblogin();
DBSETLUSER(login, "user");
DBSETLPWD(login, "my_passwd");
DBSETLAPP(login, "example2");
q_dbproc = dbopen(login, "my_server");
u_dbproc = dbopen(login, "my_server");
```

```
dbcmd(q_dbproc, "SELECT abstract FROM articles WHERE article_i
dbsqlexec(q_dbproc);
dbresults(q_dbproc);
dbbind(q_dbproc, 1, STRINGBIND, (DBINT)0, part1);
```

while (dbnextrow(q_dbproc) != NO_MORE_ROWS)
{
 // Change the value of part of the text column. This example
 // adds a sentence to the end of the existing text.
 // Update the text column.

```
dbwritetext(u_dbproc, "articles.abstract", dbtxptr(q_dbproc, 1),
 (DBINT)(strlen(part1) + strlen(part2)), NULL);
dbsqlok(u_dbproc);
dbresults(u_dbproc);
```

// Send the update value in chunks.

```
dbmoretext(u_dbproc, (DBINT)strlen(part1), part1);
dbmoretext(u_dbproc, (DBINT)strlen(part2), part2);
dbsqlok(u_dbproc);
while (dbresults(u_dbproc) != NO_MORE_RESULTS);
}
dbexit();
```

Note Notice the required calls to **dbsqlok** and **dbresults**, between the call to **dbwritetext** and the first call to **dbmoretext** and after the final call to **dbmoretext**.

When **dbwritetext** is used with **dbmoretext**, it locks the specified database **text** column. The lock is not released until the final **dbmoretext** has sent its data. This ensures that a second application does not read or update the **text** column in the midst of the first application's update.

See Also

dbmoretext dbtxptr dbreadtext dbtxtimestamp dbresults dbtxtsnewval dbsqlok dbtxtsput

Browse Functions

DB-Library for C version 7.0 supports several browse functions.

dbcolbrowse

Determines whether the source of a result column can be updated with the DB-Library browse-mode facilities.

Syntax

BOOL dbcolbrowse (PDBPROCESS *dbproc*, **INT** *colnum*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

colnum

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

Returns

TRUE or FALSE.

Remarks

The **dbcolbrowse** function is one of the DB-Library browse-mode functions. It determines whether the database column that is the source of a result column in a select list can be updated with the DB-Library browse-mode facilities. This function is useful for examining ad hoc queries. If the query has been hard-coded into the program, **dbcolbrowse** is unnecessary.

The **dbcolbrowse** function can be called any time after **dbresults**.

To determine the name of the source column given the name of the result column, use **dbcolsource**.

Examples

Only a column derived from a table that has a unique index and a **timestamp** column can be updated. It 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
<u>dbqual</u>
dbtsnewlen
dbtabbrowse
dbtsnewval
<u>dbtabcount</u>
<u>dbtsput</u>
<u>dbtabname</u>
Programming with DB-Library for C
<u>dbtabsource</u>
Browse Mode

dbcolsource

Returns a pointer to the name of the database column from which the specified result column was derived.

Syntax

LPCSTR dbcolsource (PDBPROCESS dbproc, INT colnum);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

colnum

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

Returns

A pointer to a null-terminated column name. This pointer is NULL 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

The **dbcolsource** function is one of the DB-Library browse-mode functions. This function gives an application the information it needs to update a database column, based on an ad hoc query. SELECT statements can optionally specify header names for result columns. For example:

SELECT author = au_lname FROM authors FOR BROWSE

When updating a table, you must use the database column name, not the header name (in this example, *au_lname*, not *author*). You can use the **dbcolsource** function if FOR BROWSE is issued on a view and to get the underlying database column name. For example:

```
dbcolsource(dbproc, 1)
```

This call returns a pointer to the string *au_lname*.

The **dbcolsource** function is useful for ad hoc queries. If the query has been hardcoded into the program, this function is unnecessary.

The **dbcolsource** function can be called any time after **dbresults** returns SUCCEED.

See Also

<u>dbqual</u>

<u>dbtsnewlen</u>

<u>dbresults</u>

<u>dbtsnewval</u>

<u>dbtabbrowse</u>

<u>dbtsput</u>

<u>dbtabcount</u>

Programming with DB-Library for C

<u>dbtabname</u>

Browse Mode

<u>dbtabsource</u>

dbfreequal

Frees memory allocated by the **dbqual** function.

Syntax

```
void dbfreequal ( LPCSTR ptr );
```

Arguments

ptr

Is a pointer to the memory allocated by the **dbqual** function.

Remarks

Whenever a successful call to **dbqual** has been issued, a buffer is dynamically allocated by **dbqual**. The buffer remains allocated to the application. When the application is finished using the buffer, call **dbfreequal** to free it.

dbqual

Returns a pointer to a WHERE clause to update the current row in a browsable table.

Syntax

LPCSTR dbqual (PDBPROCESS dbproc, INT tabnum, LPCSTR tabname);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

tabnum

Is the number of the table, as specified in the SELECT statement FROM clause. Table numbers start at 1. If *tabnum* is -1, the *tabname* parameter is used to identify the table.

tabname

Is a pointer to the null-terminated name of a table specified in the SELECT statement FROM clause. If *tabname* is NULL, the *tabnum* parameter is used to identify the table.

Returns

A pointer to a null-terminated WHERE clause for the current row in the specified table. This buffer is dynamically allocated, and the application must free it by calling **dbfreequal**. If **dbqual** is asked to construct a WHERE clause for a table that cannot be browsed, it returns a NULL pointer. For details, see **dbtabbrowse**.
Remarks

The **dbqual** function is one of the DB-Library browse-mode functions.

The **dbqual** function provides a WHERE clause that the application can use 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 **dbqual** begins with the keyword WHERE and contains references to the row's unique index and **timestamp** column. The application appends the WHERE clause to an UPDATE or DELETE statement without needing to examine or manipulate the statement in any way.

The **timestamp** column indicates the time that the particular row was last updated. An update on a browsable table fails if the **timestamp** column in the **dbqual**-generated WHERE clause is different from the **timestamp** column in the table. Such a condition, which provokes SQL Server error message 532, indicates that another user updated the row between the time this application selected it for browsing and the time it tried to update it. The application itself must provide the logic for handling the update failure. For one approach, see the example in **dbtabbrowse**.

The **dbqual** function can construct WHERE clauses only for tables that can be browsed. Use **dbtabbrowse** to determine whether a table can be browsed.

The **dbqual** function is usually called after **dbnextrow**.

Examples

This code fragment shows what to do when a browse-mode update fails because another user has already updated the row. This example retrieves the entire row again, allows the user to examine and modify it, and tries the update again.

Note that q_dbproc is the DBPROCESS used to query the database, and u_dbproc is the DBPROCESS used to update the database.

```
// First, find out which employee record the user wants to update.
employee_id = which_employee();
while (1)
{
    // Retrieve that employee record from the database.
```

// Assume that "empid" is a unique index, so this query will
// return only one row.

```
dbfcmd
(q_dbproc,
''SELECT * FROM employees WHERE empid = %d FOR BROWSI
employee_id);
dbsqlexec(q_dbproc);
dbresults(q_dbproc);
while (dbnextrow(q_dbproc)
    != NO_MORE_ROWS);
```

// Now, let the user examine or edit the employee's
// data, first placing the data into the program variables.

```
extract_employee_data(q_dbproc, employee_struct);
examine_and_edit(employee_struct, &edit_flag);
```

```
if (edit_flag == FALSE)
{
    // The user didn't edit this record,
    // so it's done.
    break;
}
else
{
    // The user edited this record, so the application
    // uses the edited data to update the corresponding row
    // in the database.
    qualptr = dbqual(q_dbproc, -1, "employees");
```

```
dbcmd(u_dbproc, "update employees");
```

```
dbfcmd
(u_dbproc,
" SET address = '%s', salary = %d %s",
employee_struct->address, employee_struct->salary,
qualptr);
dbfreequal(qualptr);
if ((dbsqlexec(u_dbproc) == FAIL)
|| (dbresults(u_dbproc) == FAIL))
{
 // The update failed. In a real program, it
 // would be necessary to examine the messages
 // returned from SQL Server to determine
 // why it failed. In this example,
 // assume that the update failed because
 // someone else has already updated this
 // row, thereby changing the timestamp.
 //
 // To cope with this situation, repeat
 // the loop, retrieving the changed row
 // for the user to examine and edit.
 // This will give the user the opportunity
 // to decide whether to overwrite
 // the change made by the other user.
 continue;
}
else
ł
 // The update succeeded, so the
 // application is done.
 break;
```

```
}
```

} }

See Also

- <u>dbcolbrowse</u>
- <u>dbtabname</u>

<u>dbcolsource</u>

<u>dbtabsource</u>

<u>dbnextrow</u>

<u>dbtsnewlen</u>

<u>dbtabbrowse</u>

<u>dbtsnewval</u>

<u>dbtabcount</u>

<u>dbtsput</u>

dbtabbrowse

Determines whether the specified table can be updated with the DB-Library browse-mode facilities.

Syntax

BOOL dbtabbrowse (PDBPROCESS *dbproc*, **INT** *tabnum*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

tabnum

Is the number of the table as specified in the SELECT statement's FROM clause. Table numbers start at 1.

Returns

TRUE or FALSE. If you drop the unique index of a table while browsing, **dbtabbrowse** continues to return TRUE.

Remarks

The **dbtabbrowse** function is one of the DB-Library browse-mode functions. For a detailed discussion of browse mode, see Browse Mode in Programming DB-Library for C.

The **dbtabbrowse** function provides a way to identify tables that can be browsed. It is useful for examining ad hoc queries prior to performing browsemode updates based on the queries. If the query has been hard-coded into the program, this function is unnecessary. A table must have a unique index and a **timestamp** column before it can be browsed.

Call **dbtabbrowse** any time after calling **dbresults**.

See Also

dbcolbrowsedbtabsourcedbcolsourcedbtsnewlendbtsnewlendbtsnewlendbtsnewvaldbtsnewvaldbtabnamedbtsput

dbtabcount

Returns the number of tables involved in the current SELECT statement.

Syntax

INT dbtabcount (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The number of tables, including SQL Server work tables, involved in the current set of row results. A value of -1 is returned if an invalid *dbproc* value is sent to **dbtabcount**.

Remarks

The **dbtabcount** function is one of the DB-Library browse-mode functions. For a detailed discussion of browse mode, see Browse Mode in Programming DB-Library for C.

A SELECT statement can generate a set of result rows whose columns are derived from several database tables. To perform browse-mode updates of columns in a statement's select list, the application must know how many tables were involved in the query, because each table requires a separate UPDATE statement. The **dbtabcount** function can provide this information for ad hoc queries. If the query has been hard-coded into the program, this function is unnecessary.

The count returned by this function includes any SQL Server work tables used in

processing the query. The SQL Server sometimes creates temporary, internal work tables to process a query. It deletes these work tables by the time it finishes processing the statement. Work tables cannot be updated and are not available to the application. Therefore, before using a table number, the application must make sure that it does not belong to a work table. The **dbtabname** function can be used to determine whether a particular table number refers to a work table.

The **dbtabcount** function can be called any time after **dbresults**.

See Also

dbcolbrowsedbtabsourcedbcolsourcedbtsnewlendbqualdbtsnewvaldbresultsdbtsput

dbtabname

Returns the name of a table based on its number.

Syntax

LPCSTR dbtabname (PDBPROCESS dbproc, INT tabnum);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

tabnum

Is the number of the table. Table numbers start with 1. Use **dbtabcount** to find out the total number of tables involved in a particular statement.

Returns

A pointer to the null-terminated name of the specified table. This pointer is NULL 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 **dbtabcount**.

Remarks

dbtabname is one of the DB-Library browse-mode functions. For a detailed discussion of browse mode, see Browse Mode in Programming DB-Library for C.

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. **dbtabname** provides a way for an application to determine the name of each table involved in an ad hoc query. If the query has been hard-coded into the program, this function is unnecessary.**dbtabname** can be called any time after **dbresults**.

See Also

dbcolbrowsedbtabbrowsedbcolsourcedbtsnewlendbqualdbtsnewvaldbresultsdbtsput

dbtabsource

Returns the name and number of the table from which a particular result column was derived.

Syntax

LPCSTR dbtabsource (PDBPROCESS dbproc, INT colnum, LPINT tabnum);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

colnum

Is the number of the result column. Column numbers start at 1.

tabnum

Is a pointer to an integer, which is filled in with the table's number. Many DB-Library functions that deal with browse mode accept either a table name or a table number. If **dbtabsource** returns NULL (see Returns), *tabnum* is set to -1.

Returns

A pointer to the name of the table from which this result column was derived. A NULL return value can mean one of the following:

• The DBPROCESS structure is inactive or not enabled. This is an error that causes an application's error handler to be invoked.

- The SELECT statement does not contain the FOR BROWSE clause.
- The column number is out of range.
- The column is the result of an expression, such as "MAX(*colname*)".

Remarks

dbtabsource is one of the DB-Library browse-mode functions. For a detailed discussion of browse mode, see Browse Mode in Programming DB-Library for C.

The **dbtabsource** function allows an application to determine which tables provided the columns in the current set of result rows. This information is valuable when using **dbqual** to construct WHERE clauses for UPDATE and DELETE statements based on ad hoc queries. If the query has been hard-coded into the program, this function is unnecessary.

The **dbtabsource** function can be called any time after **dbresults**.

See Also dbcolbrowse dbtabbrowse dbcolsource dbtabcount dbtabcount dbqual dbtsnewval dbresults

dbtsnewlen

Returns the length of the new value of the **timestamp** column after a browsemode update.

Syntax

INT dbtsnewlen (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The length, in bytes, of the new **timestamp** value for the updated row. If no timestamp was returned to the application, the function returns -1.

Remarks

dbtsnewlen is one of the DB-Library browse-mode functions. For a detailed discussion of browse mode, see Browse Mode in <u>Programming with DB-Library</u> <u>for C</u>.

dbtsnewlen provides information about the **timestamp** column. **dbtsnewlen** allows the application to save the length of the new **timestamp** value, possibly for use with **dbtsput**.

See Also

<u>dbcolbrowse</u> <u>dbtabcount</u> <u>dbcolsource</u>

<u>dbtabname</u>

<u>dbqual</u>

<u>dbtsput</u>

<u>dbtabbrowse</u>

dbtsnewval

Returns the new value of the **timestamp** column after a browse-mode update.

Syntax

LPCDBBINARY dbtsnewval (PDBPROCESS dbproc);

Argument

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

A pointer to the new **timestamp** value for the updated row. If no timestamp was returned to the application (possibly because the update was unsuccessful), the pointer is NULL.

IMPORTANT Do not modify the identifier in any way. Modifying the identifier can cause unpredictable results.

Remarks

dbtsnewval is one of the DB-Library browse-mode functions.

dbtsnewval provides information about the **timestamp** column. This function allows the application to save the new **timestamp** value, possibly for use with **dbtsput**.

See Also

<u>dbcolbrowse</u>

<u>dbtabcount</u>

<u>dbcolsource</u>

<u>dbtabname</u>

<u>dbqual</u>

<u>dbtabsource</u>

<u>dbtabbrowse</u>

dbtsput

Puts the new value of the **timestamp** column into the given table's current row in the DBPROCESS structure.

Syntax

RETCODE dbtsput (PDBPROCESS dbproc, LPCDBBINARY newts, INT newtslen, INT tabnum, LPCSTR tabname);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server. This must be the DBPROCESS used to perform the original SELECT query.

newts

Is a pointer to the new **timestamp** value. It is returned by **dbtsnewval**.

newtslen

Is the length of the new **timestamp** value. It is returned by **dbtsnewlen**.

tabnum

Is the number of the updated table. Table numbers start at 1. The *tabnum* must refer to a table that can be browsed. The **dbtabbrowse** function determines whether a table can be browsed. If this value is -1, the *tabname* parameter identifies the table.

tabname

Is a pointer to a null-terminated table name. The *tabname* must refer to a

table that can be browsed. Set this pointer to NULL if the *tabnum* parameter is used to identify the table. The value of *tabname* is returned by **dbtabsource**.

Returns

SUCCEED or FAIL. The following situations cause this function to return FAIL:

- The application tries to update the timestamp of a nonexistent row.
- The application tries to update the timestamp, using NULL as the new **timestamp** value (*newts*).
- The specified table cannot be browsed.

Remarks

The **dbtsput** function is one of the DB-Library browse-mode functions. For a detailed discussion of browse mode, see Browse Mode in Programming DB-Library for C.

dbtsput manipulates the **timestamp** column. If the same row is updated a second time, the UPDATE statement's WHERE clause must use the latest **timestamp** value. This function updates the timestamp in the DBPROCESS for the row currently being browsed. Then, if the application needs to update the row a second time, it can call **dbqual** to formulate a new WHERE clause that uses the new timestamp. Call **dbtsput** only after **dbnextrow** or dbgetrow has returned REG_ROW.

See Also

dbcolbrowse dbtabcount dbcolsource

dbtabname

<u>dbqual</u> <u>dbtabsource</u>

<u>dbtabbrowse</u>

<u>dbtsnewlen</u>

Bulk-Copy Functions

DB-Library for C version 7.0 supports several bulk-copy functions.

bcp_batch

Saves any rows previously bulk copied from program variables and sent to Microsoft® SQL Server[™] 2000 by **bcp_sendrow**.

Syntax

DBINT bcp_batch (PDBPROCESS dbproc);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The number of rows saved after the last call to **bcp_batch**, or **-**1 in case of error.

Remarks

When an application uses **bcp_bind** and **bcp_sendrow** to bulk copy rows from program variables to SQL Server tables, the rows are permanently saved in SQL Server only when the program calls **bcp_batch** or **bcp_done**.

You can call **bcp_batch** once every *n* rows or when there is a lull between periods of incoming data (as in a telemetry application). Of course, you can choose some other criteria, or you can decide not to call **bcp_batch** at all. If **bcp_batch** is not called, the rows are permanently saved in SQL Server when **bcp_done** is called.

See Also

bcp_done

bcp_sendrow

bcp_bind

Binds data from a program variable to a table column for bulk copy into Microsoft® SQL Server[™] 2000.

Syntax

RETCODE bcp_bind (PDBPROCESS dbproc, LPCBYTE varaddr, INT prefixlen, DBINT varlen, LPCBYTE terminator, INT termlen, INT type, INT table_column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

varaddr

Is the address of the program variable from which the data is copied. If *type* is SQLTEXT or SQLIMAGE, *varaddr* can be NULL. A NULL *varaddr* indicates that **text** and **image** values are sent to SQL Server in chunks by **bcp_moretext**, rather than all at once by **bcp_sendrow**.

prefixlen

Is the length, in bytes, of any length prefix this column can have; valid length prefixes are 0, 1, 2, or 4 bytes. For example, strings in some non-C programming languages are made up of a 1-byte length prefix, followed by the string data itself. If the data does not have a length prefix, set *prefixlen* to

varlen

Is the length of the data in the program variable, not including the length of any length prefix or terminator. Setting *varlen* to 0 signifies that the data is NULL. Setting *varlen* to -1 indicates that the system should ignore this parameter.

For fixed-length data types, such as integers, the data type itself indicates to the system the length of the data. Therefore, for fixed-length data types, *varlen* must always be -1 except when the data is NULL, in which case *varlen* must be 0.

For character, **text**, **binary**, and **image** data, *varlen* can be -1, 0, or some positive value. If *varlen* is -1, the system uses either a length prefix or a terminator sequence to determine the length. (If both are supplied, the system uses the one that results in the shortest amount of data being copied.) If *varlen* is -1 and neither a prefix length nor a terminator sequence is specified, the system returns an error message. If *varlen* is 0, the system assumes the data is NULL. If *varlen* is some positive value, the system uses *varlen* as the data length. However, if, in addition to a positive *varlen*, a prefix length or terminator sequence is provided, the system determines the data length by using the method that results in the shortest amount of data being copied.

terminator

Is a pointer to the byte pattern, if any, that marks the end of this program variable. For example, C strings usually have a 1-byte terminator whose value is 0. If there is no terminator for the variable, set *terminator* to NULL. If you want to designate the C null terminator as the program variable terminator, use an empty string (" ") as *terminator* and set *termlen* to 1, because the null terminator constitutes a single byte. For instance, to use a C null terminator:

bcp_bind (dbproc, co_name, 0, -1, "", 1, 0, 2)

If there is no terminator:

bcp_bind (dbproc, co_name, 0, -1, NULL, 0, 0, 2)

0.

termlen

Is the length of this program variable's terminator, if any. If there is no terminator for the variable, set *termlen* to 0.

type

Is the data type of your program variable. The data in the program variable is automatically converted to the type of the database column. If this parameter is 0, no conversion is performed. For more information about a list of supported conversions, see **dbconvert**. For the list of valid SQL Server data types, see <u>DB-Library for C Data Types</u>.

table_column

Is the column number in the database table to which the data is copied. Column numbers start at 1.

Returns

SUCCEED or FAIL.

Remarks

Use **bcp_bind** for a fast, efficient way to copy data from a program variable into a table in SQL Server without first putting the data into a user file or using the Transact-SQL INSERT statement.

Call **bcp_init** before calling this or any other bulk-copy functions.

Make a separate **bcp_bind** call for every column in the SQL Server table into which you want to copy. After the necessary **bcp_bind** calls have been made, then call **bcp_sendrow** to send a row of data from your program variables to SQL Server. Call **bcp_init** to set the table to be copied into.

Whenever you want SQL Server to checkpoint the rows already received, call **bcp_batch**. For example, call **bcp_batch** once for every 1000 rows inserted or at any other interval.

When there are no more rows to be inserted, call **bcp_done**. Failure to do so results in an error.

When using **bcp_bind**, the user file name parameter, set *hfile*, in the call to

bcp_init, to NULL, and set *direction*, the direction parameter, to DB_IN.

Control parameter settings, specified with **bcp_control**, have no effect on **bcp_bind** row transfers.

Calling **bcp_columns** when using **bcp_bind** results in an error.

Examples

The following example shows how to use **bcp_bind**:

```
LOGINREC *login;
DBPROCESS *dbproc;
       co name[MAXNAME];
char
          co id;
DBINT
DBINT
          rows sent;
DBBOOL
            more_data;
       *terminator = "\t\t";
char
// Install error-handler and message-handler.
dberrhandle(err_handler);
dbmsghandle(msg handler);
// Open a DBPROCESS structure.
login = dblogin();
DBSETLUSER(login, "user");
DBSETLPWD(login, "my_passwd");
DBSETLAPP(login, "example");
BCP SETL(login, TRUE);
dbproc = dbopen(login, "my_server");
// Initialize bulk copy.
if (bcp_init(dbproc, "comdb..accounts_info", (BYTE *)NULL,
 (BYTE *)NULL, DB_IN) == FAIL)
 exit(ERREXIT);
```

// Bind program variables to table columns.
if (bcp_bind(dbproc, (BYTE *)&co_id, 0, (DBINT)-1, (BYTE *)NUL)
{
```
fprintf(stderr, "bcp_bind, column 1, failed.\n");
 exit(ERREXIT);
}
if (bcp_bind
 (dbproc, co_name, 0, (DBINT)-1, terminator, strlen(terminator), 0,
{
 fprintf(stderr, "bcp_bind, column 2, failed.\n");
 exit(ERREXIT);
}
while (TRUE)
{
 // Process and retrieve program data.
 more_data = getdata(&co_id, co_name);
 if (more_data == FALSE)
   break:
   // Send the data.
 if (bcp_sendrow(dbproc) == FAIL)
   exit(ERREXIT);
}
// Terminate the bulk copy operation.
if ((rows_sent = bcp_done(dbproc)) == -1)
 printf("Bulk copy unsuccessful.\n");
else
 printf("%ld rows copied.\n", rows_sent);
```

See Also

bcp_batch bcp_exec bcp_collen bcp_init

bcp_colptr

bcp_moretext

<u>bcp_columns</u>

bcp_sendrow

<u>bcp_control</u>

<u>dbconvert</u>

bcp_done

DB-Library for C Data Types

bcp_colfmt

Specifies the format of a user file for bulk copy into or out of Microsoft® SQL Server[™] 2000.

Syntax

RETCODE bcp_colfmt (PDBPROCESS *dbproc*,

INT file_column, BYTE file_type, INT file_ prefixlen, DBINT file_collen, LPCBYTE file_term, INT file_termlen, INT table_column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or

SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

file_column

Is the column number in the user file for which the format is being specified. The first column is number 1.

file_type

Is the data type of this column in the user file. If it is different from the data type of the corresponding column in the database table (*table_column*), the conversion is performed automatically. For a table of allowable data conversions, see <u>dbconvert</u>. If you want to specify the same data type as in the corresponding column of the database table (*table_column*), set this parameter to 0.

For a bulk copy out of SQL Server into a file, when *file_type* 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** and **numeric**, the precision and scale of the source column are used.

file_prefixlen

Is the length of the length prefix for this column in the user file. Legal prefix lengths are 1, 2, and 4 bytes. To avoid using a length prefix, set this parameter to 0. To let DB-Library determine whether to use a length prefix, set this parameter to -1. In such a 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 a user-file column length is used (such as a length prefix and a maximum column length, or a length prefix and a terminator sequence), DB-Library chooses the one that results in the shortest amount of data being copied.

Length prefixes simplify the way null data values are specified in a user file. For instance, assume 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 for the column.

file_collen

Is the maximum length of this column's data in the user file, not including the length of any length prefix or terminator. Setting *file_collen* to 0 signifies that the data is NULL. Setting *file_collen* to -1 indicates that the system should ignore this parameter (that is, there is no default maximum length).

For fixed-length data types, such as integers, the length of the data is constant, except for the special case of null values. Therefore, for fixed-length data types, *file_collen* must always be -1, except when the data is NULL, in which case *file_collen* must be 0. For character, **text**, **binary**, and **image** data, *file_collen* can be -1, 0, or some positive value. If *file_collen* 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 shortest amount of data being copied.) If *file_collen* is -1 and neither a prefix length nor a terminator sequence is specified, the system returns an error message. If *file_collen* is 0, the system assumes the data is NULL. If *file_collen* is a positive value, the system uses *file_collen* as the maximum data length. However, if, in addition to a positive *file_collen*, a prefix length or terminator sequence is provided, the system determines the data length by using the method that results in the shortest amount of data being copied.

file_term

Is the terminator sequence to be used for this column. This parameter is useful mainly for character, **text**, **binary**, and **image** data types 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 *file_term* to "\0". To make the tab character the terminator, set *file_term* to "\t". To make the newline character the terminator, set *file_term* to "\n".

If more than one means of specifying a user-file column length is used (such as a terminator and a length prefix, or a terminator and a maximum column length), the bulk copy chooses the one that results in the shortest amount of data being copied.

file_termlen

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.

table_column

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 or FAIL.

Remarks

bcp_colfmt allows you to specify the user-file format for bulk copies. For bulk copy, a format contains the following parts:

- A mapping from user-file columns to database columns.
- The data type of each user-file column.
- The length of the optional length prefix of each column.
- The maximum length of the user-file column's data.
- The optional terminating byte sequence for each column.
- The length of this optional terminating byte sequence.

Each call to **bcp_colfmt** specifies the format for one user-file column. For example, to change the default settings for three columns in a five-column table, first call **bcp_columns(5)**, and then call **bcp_colfmt** five times, with three of those calls setting your custom format. Set *file_type* of the remaining two calls to 0, and set their *file_prefixlen, file_collen,* and *file_termlen* parameters to -1. This procedure copies all five columns, three with your customized format and two with the default format.

The **bcp_columns** function must be called before any calls to **bcp_colfmt**.

You must call **bcp_colfmt** once and only once for every column in the user file, regardless of whether some of those columns use the default format or are skipped.

To skip a column, set the *table_column* parameter to 0. If you want to skip a column, you must specify its type.

See Also

<u>bcp_batch</u>

<u>bcp_exec</u>

bcp_colptr

<u>bcp_init</u>

<u>bcp_columns</u>

bcp_sendrow

bcp_control

<u>dbconvert</u>

bcp_done

bcp_collen

Sets the program variable data length for the current bulk copy into Microsoft® SQL Server[™] 2000.

Syntax

RETCODE bcp_collen (PDBPROCESS dbproc, DBINT varlen, INT table_column);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

varlen

Is the length of the program variable, which does not include the length of the length prefix or terminator. Setting *varlen* to 0 signifies that the data is NULL. Setting it to -1 signifies that the data is variable length and that the length is determined by the length prefix or terminator. If both a length prefix and a terminator exist, **bcp** uses the one that results in the shortest amount of data being copied.

table_column

Is the column in the SQL Server table to which the data is copied. Column numbers start at 1.

Returns

SUCCEED or FAIL.

Remarks

The **bcp_collen** function allows you to change the program variable data length for a particular column while running a copy in through calls to **bcp_bind**.

Initially, the program variable data length is determined when **bcp_bind** is called. If the program variable data length changes between calls to **bcp_sendrow** and no length prefix or terminator is being used, you can call **bcp_collen** to reset the length. The next call to **bcp_sendrow** uses the length you just set.

There must be a separate **bcp_collen** call for every column in the table whose data length you want to modify.

See Also

bcp_bind

bcp_sendrow

bcp_colptr

Sets the program variable data address for the current copy into Microsoft® SQL Server[™] 2000.

Syntax

RETCODE bcp_colptr (PDBPROCESS *dbproc*, **LPCBYTE** *colptr*, **INT** *table_column* **);**

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

colptr

Is the address of the program variable.

table_column

Is the column in the SQL Server table to which the data is copied. Column numbers start at 1.

Returns

SUCCEED or FAIL.

Remarks

The **bcp_colptr** function allows you to change the program variable data address for a particular column while running a copy in through calls to **bcp_bind**.

Initially, the program variable data address is determined when **bcp_bind** is called. If the program variable data address changes between calls to

bcp_sendrow, you can call **bcp_colptr** to reset the address of the data. The next call to **bcp_sendrow** uses the data at the address you just set.

There must be a separate **bcp_colptr** call for every column in the table whose data address you want to modify.

See Also

bcp_bind

bcp_sendrow

bcp_columns

Sets the total number of columns found in the user file for use with a bulk copy into or out of Microsoft® SQL Server[™] 2000.

Syntax

RETCODE bcp_columns (PDBPROCESS *dbproc*, **INT** *file_colcount* **)**;

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

file_colcount

Is the total number of columns in the user file. Even if you are preparing to bulk copy data from the user file to a SQL Server table and do not intend to copy all columns in the user file, you must still set *file_colcount* to the total number of user-file columns.

Returns

SUCCEED or FAIL.

Remarks

This function can be called only after **bcp_init** has been called with a valid file name.

You should call this function only if you intend to use a user-file format that differs from the default. For a description of the default user-file format, see **bcp_init**.

After calling **bcp_columns**, you must call **bcp_colfmt** *file_colcount* times because you are defining a completely custom file format.

See Also

bcp_colfmt

bcp_init

bcp_control

Changes the default settings for various control parameters for a bulk copy between a file and Microsoft[®] SQL Server[™] 2000.

Syntax

RETCODE bcp_control (PDBPROCESS dbproc, INT field, DBINT value);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

field

One of the following:

BCPMAXERRS

Is the number of errors allowed before giving up. The default is 10; a value less than 1 resets this field to its default value. If a value larger than 65,535 is specified, this field is set to 65,535.

BCPFIRST

Is the first row to copy. The default is 1; a value less than 1 resets this field to its default value.

BCPLAST

Is the last row to copy. The default is to copy all rows; a value less than 1 resets this field to its default value.

BCPBATCH

Is the number of rows per batch. The default is 0; a value less than 1 resets this field 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 **bcp_exec**, 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 *field*.

Returns

SUCCEED or FAIL.

Remarks

This function sets various control parameters for bulk-copy operations, including the number of errors allowed before canceling a bulk copy, the numbers of the first and last rows to copy, and the batch size.

These control parameters are only meaningful when copying between a user file and a SQL Server table. Control parameter settings have no effect on **bcp_bind** row transfers.

The **bcp_control** function has a *field* parameter: BCPABORT. BCPABORT stops a bulk-copy operation that is already in progress. Call **bcp_control** with a *field* of BCPABORT from another thread to stop a running bulk-copy operation. The *value* parameter is ignored.

You can pass hints string to **bcp_control**. When performing a bulk copy operation, the TABLOCK hint can be specified using **bcp_control** to acquire a table lock instead of row locks.

Examples

The following example shows how to use **bcp_control**:

LOGINREC *login; DBPROCESS *dbproc; DBINT rowsread;

// Install error-handler and message-handler.
dberrhandle(err_handler);
dbmsghandle(msg_handler);

```
// Open a DBPROCESS structure.
login = dblogin();
BCP_SETL(login, TRUE);
DBSETLUSER(login, "user");
DBSETLPWD(login, "my_passwd");
DBSETLAPP(login, "example");
dbproc = dbopen(login, "my_server");
```

```
// Initialize bulk copy.
if (bcp_init(dbproc, "comdb..address", "address.add", "addr.err",
    DB_IN) == FAIL)
    exit(ERREXIT);
```

```
// Set the number of rows per batch.
if (bcp_control(dbproc, BCPBATCH, (DBINT) 1000) == FAIL)
{
    printf("bcp_control failed to set batching behavior.\n");
    exit(ERREXIT);
}
// Set file column count.
if (bcp_columns(dbproc, 1) == FAIL)
{
    printf("bcp_columns failed.\n");
```

```
exit(ERREXIT);
}
// Set the file format.
if (bcp_colfmt(dbproc, 1, 0, 0, (DBINT)-1, "\n", 1, 1) == FAIL)
{
    printf("bcp_colformat failed.\n");
    exit(ERREXIT);
}
// Now, execute the bulk copy.
if (bcp_exec(dbproc, &rowsread) == FAIL)
{
    printf("Incomplete bulk copy. Only %ld row%c copied.\n",
    rowsread, (rowsread == 1) ? ' ': 's');
    exit(ERREXIT);
}
```

See Also

<u>bcp_batch</u>

<u>bcp_colptr</u>

bcp_bind

<u>bcp_exec</u>

<u>bcp_colfmt</u>

bcp_init

bcp_collen

bcp_done

Ends a bulk copy from program variables into Microsoft® SQL Server[™] 2000 performed with **bcp_bind** and **bcp_sendrow**.

Syntax

DBINT bcp_done (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

The number of rows permanently saved after the last call to **bcp_batch**, or **-**1 in case of error.

Remarks

Call **bcp_done** after the last call to **bcp_sendrow** or **bcp_moretext**. Failure to call **bcp_done** after copying in all your data results in unpredictable errors.

See Also

bcp_batch

bcp_moretext

bcp_bind

bcp_sendrow

bcp_exec

Executes a complete bulk copy of data between a database table and a user file.

Syntax

RETCODE bcp_exec (PDBPROCESS *dbproc*, **LPDBINT** *rows_copied* **);**

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

rows_copied

Is a pointer to a DBINT. The **bcp_exec** function fills this DBINT with the number of rows successfully copied. If set to NULL, this parameter is not filled in by **bcp_exec**.

Returns

SUCCEED or FAIL. The **bcp_exec** function returns SUCCEED if all rows are copied. If a partial or complete failure occurs, **bcp_exec** returns FAIL. Check the *rows_copied* parameter for the number of rows successfully copied.

Remarks

This function copies data from a user file to a database table or vice versa, depending on the value of the *direction* parameter in **bcp_init**.

Before calling **bcp_exec**, call **bcp_init** with a valid user file name. Failure to do so results in an error.

Examples

The following example shows how to use **bcp_exec**:

LOGINREC *login; DBPROCESS *dbproc; DBINT rowsread;

// Install error-handler and message-handler.
dberrhandle(err_handler);
dbmsghandle(msg_handler);

```
// Open a DBPROCESS structure.
login = dblogin();
DBSETLUSER(login, "user");
DBSETLPWD(login, "my_passwd");
DBSETLAPP(login, "example");
BCP_SETL(login, TRUE);
dbproc = dbopen(login, "my_server");
```

// Initialize bulk copy.
if (bcp_init(dbproc, "pubs..authors", "authors.sav",
 (BYTE *)NULL, DB_OUT) == FAIL)
 exit(ERREXIT);

// Now, execute the bulk copy.

if (bcp_exec(dbproc, &rowsread) == FAIL)
printf("Incomplete bulk copy. Only %ld row%s copied.\n",
rowsread, (rowsread == 1) ? "": "s");

See Also

bcp_batch

bcp_colptr

<u>bcp_bind</u>

<u>bcp_columns</u>

bcp_colfmt

bcp_control

<u>bcp_collen</u>

bcp_sendrow

bcp_init

Initializes bulk copy operation.

Syntax

RETCODE bcp_init (PDBPROCESS dbproc, LPCSTR tblname, LPCSTR hfile, LPCSTR errfile, INT direction);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or Microsoft® SQL Server[™] 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

tblname

Is the name of the database table to be copied in or out. This name can also include the database name or the database owner name. For example, **pubs.gracie.titles**, **pubs..titles**, **gracie.titles**, and **titles** are all valid table names.

If *direction* is DB_OUT, *tblname* can also be the name of a database view.

hfile

Is the name of the user file to be copied into or out of SQL Server. If data is being copied directly from variables using **bcp_sendrow**, set *hfile* to NULL.

errfile

Is the name of the error file to be used. This error file is filled with progress messages, error messages, and copies of any rows that, for any reason, could

not be copied from a user file to a SQL Server table. If NULL is passed as *errfile*, no error file is used.

direction

Is the direction of the copy. It must be one of two values: DB_IN or DB_OUT. DB_IN indicates a copy from program variables or a user file to the database table, while DB_OUT indicates a copy from the database table to the user file. It is invalid to request a bulk copy from the database table (DB_OUT) without supplying a user file name.

Returns

SUCCEED or FAIL.

Remarks

bcp_init performs the necessary initializations for a bulk copy of data between the workstation and SQL Server. It sets the default user-file data formats and examines the structure of the database table.

If a user file is being used (see the description of the *hfile* parameter), the default data native formats are as follows:

- The order, type, length, and number of the columns in the user file are assumed to be identical to the order, type, and number of the columns in the database table.
- If a given database column's data is fixed length, then the user file's data column is also fixed length.

When a given database column's data is variable length or when it can contain null values, the user file's data column is prefixed by a 4-byte length value for SQLTEXT and SQLIMAGE data types and a 1-byte length value for all other types.

• There are no terminators of any kind between user file columns.

Any of these defaults can be overridden by calling **bcp_columns** and **bcp_colfmt**.

To use the bulk copy functions to copy data to a database table, follow these examples:

- Call BCP_SETL to make the DBPROCESS structure usable for bulk copy purposes: login = dblogin(); BCP_SETL(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:
 sp_dboption 'mydb', 'select into/bulkcopy', 'true'
 GO
 USE mydb
 GO
 CHECKPOINT

If no user file is being used, it is necessary to call **bcp_bind** to specify the format and location in memory for each column's data value, and send the rows using **bcp_sendrow**

The **bcp_init** function must be called before any other bulk-copy functions. Failure to do so results in an error.

See Also

bcp_batch bcp_columns bcp_bind bcp_control bcp_colfmt bcp_done bcp_collen bcp_sendrow <u>bcp_colptr</u>

bcp_moretext

Sends part of a **text** or **image** value to Microsoft® SQL Server[™] 2000.

Syntax

RETCODE bcp_moretext (PDBPROCESS *dbproc*, **DBINT** *size*, **LPCBYTE** *text* **);**

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

size

Is the size of this particular part of the **text** or **image** value being sent to SQL Server. Sending more **text** or **image** bytes to SQL Server than were specified in the call to **bcp_bind** or **bcp_collen** results in an error.

text

Is a pointer to the **text** or **image** portion to be sent to SQL Server.

Returns

SUCCEED or FAIL.

Remarks

This function is used in conjunction with **bcp_bind** and **bcp_sendrow** to send a large SQLTEXT or SQLIMAGE value to SQL Server in a number of smaller chunks. This is particularly useful with operating systems unable to allocate extremely long data buffers.

If **bcp_bind** is called with a *type* parameter of SQLTEXT or SQLIMAGE and a nonnull *varaddr* parameter, **bcp_sendrow** sends the entire **text** or **image** data value, just as it does for all other data types. If, however, **bcp_bind** has a null *varaddr* parameter, **bcp_sendrow** returns control to the application immediately after all non**text** and non**image** columns are sent to SQL Server. The application can then call **bcp_moretext** repeatedly to send the **text** and **image** columns to SQL Server, a chunk at a time.

If you use **bcp_moretext** to send one **text** or **image** column in the row, you must also use it to send all other **text** and **image** columns in the row.

If the row contains more than one **text** or **image** column, **bcp_moretext** first sends its data to the lowest numbered (that is, leftmost) **text** or **image** column, followed by the next lowest numbered column, and so on.

An application generally calls **bcp_sendrow** and **bcp_moretext** within loops to send a number of rows of data. Here's an outline of how to do this for a table containing two **text** columns:

```
while (there are still rows to send)
{
    bcp_sendrow(...);
```

```
for (all the data in the first text column)
bcp_moretext(...);
```

```
for (all the data in the second text column)
bcp_moretext(...);
}
```

Examples

This example shows how to use **bcp_moretext** with **bcp_bind** and **bcp_sendrow**:

```
LOGINREC *login;
DBPROCESS *dbproc;
```
```
DBINT id = 5;
char *part1 = "This text value isn't long,";
char *part2 = " but it's broken up into three parts";
char *part3 = " anyhow.";
```

// Install error handler and message handler.
dberrhandle(err_handler);
dbmsghandle(msg_handler);

```
// Open a DBPROCESS structure.
login = dblogin();
BCP_SETL(login, TRUE);
DBSETLUSER(login, "user");
DBSETLPWD(login, "my_passwd");
DBSETLAPP(login, "example");
dbproc = dbopen(login, "my_server");
// Initialize bulk copy.
if (bcp_init(dbproc, "comdb..articles", (BYTE *)NULL,
(BYTE *)NULL, DB_IN) == FAIL)
exit(ERREXIT);
```

```
strlen(part2) + strlen(part3)), (BYTE *)NULL, 0,
```

```
SQLTEXT, 2) == FAIL)
```

```
{
  fprintf(stderr, "bcp_bind, column 2, failed.\n");
  exit(ERREXIT);
}
```

// Now send this row, with the text value broken into three chunks.

- if (bcp_sendrow(dbproc) == FAIL)
 exit(ERREXIT);
- if (bcp_moretext(dbproc, (DBINT)strlen(part1), part1) == FAIL)
 exit(ERREXIT);
- if (bcp_moretext(dbproc, (DBINT)strlen(part2), part2) == FAIL)
 exit(ERREXIT);
- if (bcp_moretext(dbproc, (DBINT)strlen(part3), part3) == FAIL)
 exit(ERREXIT);

// All done.
bcp_done(dbproc);
dbclose(dbproc);

See Also

bcp_bind

bcp_sendrow

<u>bcp_collen</u>

<u>dbwritetext</u>

bcp_readfmt

Reads a data file format definition from a user file for a bulk copy between a file and Microsoft® SQL Server[™] 2000.

Syntax

RETCODE bcp_readfmt (PDBPROCESS *dbproc*, **LPCSTR** *file_name*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

file_name

Is the full directory and file name specification of the file containing the format definitions.

Returns

SUCCEED or FAIL.

Remarks

After **bcp_readfmt** reads a data file format definition from a host file, it makes the appropriate calls to **bcp_columns** and **bcp_colfmt**. These calls automate the bulk copy of multiple files that share a common data format.

The bulk copy utility (**bcp**) copies a database table to or from a host file in a user-specified format, which can be saved with **bcp** in definition files for datafile format. These files can later automate the process of bulk copying files that share a common format. For more information about the **bcp** utility and definition files

for datafile format, see Transact-SQL Reference.

Applications can call **bcp_writefmt** to create files that define data file formats.

Note The format file cannot have been produced by a version of the **bcp** utility program earlier than version 4.2.

Examples

The following code fragment shows how to use **bcp_readfmt**:

```
bcp_init(dbproc, "mytable", "bcpdata", "bcperrs", DB_IN);
bcp_readfmt(dbproc, "my_fmtfile");
bcp_exec(dbproc, &rows_copied);
```

See Also

<u>bcp_colfmt</u>

bcp_writefmt

bcp_columns

bcp_sendrow

Sends a row of data from program variables to Microsoft® SQL Server[™] 2000.

Syntax

RETCODE bcp_sendrow (PDBPROCESS *dbproc*);

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

Returns

SUCCEED or FAIL.

Remarks

bcp_sendrow builds a row from program variables and sends it to SQL Server.

Before calling **bcp_sendrow**, you must make calls to **bcp_bind** to specify the program variables to be used.

If **bcp_bind** is called with a *type* parameter of SQLTEXT or SQLIMAGE and a nonnull *varaddr* parameter, **bcp_sendrow** sends the entire **text** or **image** data value, just as it does for all other data types. If, however, **bcp_bind** has a null *varaddr* parameter, **bcp_sendrow** returns control to the application immediately after all non**text** and non**image** columns are sent to SQL Server. The application can then call **bcp_moretext** repeatedly to send the **text** and **image** columns to SQL Server, a chunk at a time. For an example, see **bcp_moretext**.

After the last call to **bcp_sendrow**, you must call **bcp_done** to ensure proper internal cleanup.

When **bcp_sendrow** is used to bulk copy rows from program variables into SQL Server tables, rows are permanently saved in SQL Server only when the user calls **bcp_batch** or **bcp_done**. The user can choose to call **bcp_batch** once every *n* rows or when there is a lull between periods of incoming data. Of course, the user can choose some other criteria or decide not to call **bcp_batch** at all. If **bcp_batch** is never called, the rows are permanently saved in SQL Server when **bcp_done** is called.

See Also

bcp_batch bcp_control bcp_bind bcp_done bcp_colfmt bcp_colfmt bcp_collen bcp_init bcp_colptr

bcp_columns

BCP_SETL

Sets the LOGINREC to enable bulk-copy operations.

Syntax

RETCODE BCP_SETL (PLOGINREC loginrec, **BOOL** enable);

Arguments

loginrec

Is a pointer to a LOGINREC structure, which is passed as a parameter to **dbopen**. You can get a LOGINREC structure by calling **dblogin**.

enable

Is a Boolean value (TRUE or FALSE) that specifies whether to enable bulk copy operations for the resulting DBPROCESS. By default, DBPROCESS structures are not enabled for bulk-copy operations.

Returns

SUCCEED or FAIL.

Remarks

This function sets a field in the LOGINREC structure that tells Microsoft® SQL Server[™] 2000 that the DBPROCESS connection can be used for bulk-copy operations. For it to have any effect, it must be called before **dbopen**, the function that actually allocates the DBPROCESS structure.

In applications that allow users to make ad hoc queries, you may want to avoid calling this function to keep users from initiating a bulk-copy sequence with Transact-SQL statements. Or you may want to call it with the *enable* parameter set to FALSE. After a bulk-copy sequence has begun, it cannot be stopped with a Transact-SQL statement.

See Also

<u>bcp_init</u>

<u>dbsetlhost</u>

<u>dblogin</u>

<u>dbsetlpwd</u>

<u>dbopen</u>

<u>dbsetluser</u>

<u>dbsetlapp</u>

bcp_writefmt

Writes a data file format definition to a user file for a bulk copy between a file and Microsoft® SQL Server[™] 2000.

Syntax

RETCODE bcp_writefmt (PDBPROCESS *dbproc,* **LPCSTR** *file_name* **);**

Arguments

dbproc

Is the DBPROCESS structure that is the handle for a particular workstation or SQL Server 2000 process. It contains all the information that DB-Library uses to manage communications and data between the workstation and SQL Server.

file_name

Is the full path and file name of the file containing the format definitions.

Returns

SUCCEED or FAIL.

Remarks

The format of the data file reflects previous calls to **bcp_columns** and **bcp_colfmt**.

The bulk copy utility (**bcp**) copies a database table to or from a host file in a user-specified format, which can be saved with **bcp** in definition files that store the data file format. These files can later automate the process of bulk copying files that share a common format.

Read format-definition files with **bcp_readfmt**.

Note The file produced by **bcp_writefmt** is not compatible with versions of the

bcp utility before version 4.2.

Examples

The following example shows how to use **bcp_writefmt**:

bcp_init(dbproc, "mytable", "bcpdata", "bcperrs", DB_OUT);

bcp_columns(dbproc, 3); bcp_colfmt(dbproc, 1, SQLCHAR, 0, -1, '\t', 1, 1); bcp_colfmt(dbproc, 2, SQLCHAR, 0, -1, '\t', 1, 2); bcp_colfmt(dbproc, 3, SQLCHAR, 0, -1, '\t', 1, 3);

bcp_writefmt(dbproc, "my_fmtfile");

bcp_exec(dbproc, &rows_copied);

See Also

bcp_colfmt bcp_readfmt bcp_columns

DB-Library Options

The functions **dbsetopt** and **dbclropt** use the following constants, defined in Sqldb.h, for setting and clearing options. All options are off by default.

DBANSItoOEM | DBOEMtoANSI (Microsoft® Windows® and Microsoft Windows NT® only)

Translates, through the Windows functions **AnsiToOem** and **OemToAnsi**, all characters or text coming from or going to the server. Because Microsoft MS-DOS® uses the OEM character set and Windows uses the ANSI character set, this translation becomes necessary when international characters (character code larger than 127) are in the database.

There are two ways to convert data with transfers to and from the server:

- Call dbsetopt with DBANSItoOEM or DBOEMtoANSI
- For Windows, in the Win.ini file's [SQLSERVER] section, set AutoANSItoOEM=ON

For Windows NT, use the SQL Server Network Utility **automatic ANSI to OEM** option. This modifies the Registry under the following subtree:

HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ MSSQLServer\ Client\ DB-Lib

This also sets the **AutoAnsiToOem** entry to ON (**AutoAnsiToOem:REG_SZ:ON**). This is the default entry.

The **AutoANSItoOEM** entry controls the default conversion behavior when you connect to a server. If **AutoANSItoOEM** is set to ON, conversion is turned on in the following cases:

• ANSI clients to OEM servers (Windows and Windows NT)

• OEM clients to ANSI servers (Windows NT)

If **AutoANSItoOEM** is set to OFF, conversion is turned off for all connections.

You can override the default conversion by calling **dbsetopt** after calling **dbopen**.

Use the DBANSItoOEM option to enable conversion when connecting an ANSI client to an OEM server. Use the DBOEMtoANSI option to enable conversion when connecting an OEM client (Windows NT Console application) to an ISO server. You cannot set both options for the same DBPROCESS.

DBARITHABORT

Cancels a query during execution when an arithmetic exception occurs. If neither DBARITHABORT nor DBARITHIGNORE is set, Microsoft SQL ServerTM 2000 substitutes null values and prints a warning after the query has been executed.

DBARITHIGNORE

Substitutes (without warning) null values for selected or updated values when an arithmetic exception occurs during query execution. If neither DBARITHABORT nor DBARITHIGNORE is set, SQL Server substitutes null values and prints a warning after the query has been executed.

DBBUFFER

Buffers the result rows to access them nonsequentially with **dbgetrow**. DB-Library handles this option locally. With the option set, supply a parameter that equals the number of rows you want buffered. If you choose a negative value, the buffer is set to a default size (currently 100). A value of 1 is invalid.

Row buffering keeps a specified number of SQL Server result rows in the program's memory. Without row buffering, the result row generated by each new **dbnextrow** call overwrites the contents of the previous result row. So use row buffering for programs that need to look at result rows

nonsequentially.

When turned on, DBBUFFER reduces memory and performance because each row in the buffer must be allocated and freed individually. Therefore, write the application to turn on the DBBUFFER option only if it calls **dbgetrow**. Note that row buffering, an independent issue, has nothing to do with network buffering. For more information about row buffering, see <u>dbgetrow</u>, <u>dbnextrow</u>, and <u>dbclrbuf</u>.

DBCLIENTCURSORS

Forces the use of client cursors. When this option is set, every cursor opened with **dbcursoropen** is a client cursor.

This option can be enabled for a client running Windows by placing the line **UseClientCursors=ON** in the **[SQLSERVER]** section of Win.ini. This option can be enabled for a client running Windows NT by setting the value **UseClientCursors** to **ON** (**UseClientCursors : REG_SZ : ON**) in the following Windows NT Registry key:

HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ MSSQLServer\ Client\ DB-Lib

DBNOAUTOFREE

Clears the command buffer only by a call to **dbfreebuf**. When DBNOAUTOFREE is not set, after a call to **dbsqlexec** or **dbsqlsend**, the first call to either **dbcmd** or **dbfcmd** automatically clears the command buffer before the new text is entered.

DBNOCOUNT

Stops returning information about the number of rows affected by each Transact-SQL statement. The application can otherwise get this information by calling DBCOUNT.

DBNOEXEC

Processes the query through the compile step, but does not execute it. You

can use this option with DBSHOWPLAN.

DBOFFSET

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:

- select
- from
- table
- order
- compute
- statement
- procedure
- execute
- param

Note The value *param* refers to parameters of stored procedures.

Calls to functions such as **dbsetopt** can specify these parameters in either uppercase or lowercase. For the internal types that correspond to the offsets, see <u>dbgetoff</u>. Offsets are returned only if the batch contains no syntax errors.

DBPARSEONLY

Checks the syntax of the query and returns error messages to the host. Offsets are returned if the DBOFFSET option is set and there are no errors.

DBQUOTEDIDENT

Specifies that DB-Library will automatically place double quotation marks (") around certain object names that are called quoted identifiers. The DBQUOTEDIDENT option is disabled by default.

- Call **dbsetopt** with the DBQUOTEDIDENT option to enable the use of quoted identifiers.
- Call **dbclropt** with the DBQUOTEDIDENT option to disable the use of quoted identifiers.
- Call **dbisopt** to check the status of the DBQUOTEDIDENT option.

Note When you use DBQUOTEDIDENT, you must set *param* to NULL.

The following table lists the functions and the parameters that are affected when DBQUOTEDIDENT is enabled.

Function	Parameter(s)
bcp_init	Tblname
dbrpcinit	Rpcname
dbupdatetext	dest_object
	src_object
dbuse	Dbname
dbwritetext	Objname

When you call **dbrpcinit**, you must explicitly place quotation marks around remote procedure names that require quotation marks in the *rpcname* parameter. This example demonstrates the use of quotation marks in *rpcname*.

"My server".."My RPC"

DBROWCOUNT

Specifies a maximum number of regular rows to be returned on SELECT statements. This option does not limit the number of compute rows returned. Different from most options, DBROWCOUNT is always on, never off. Setting DBROWCOUNT to 0 sets it back to the default, returning all the rows generated by a SELECT statement. Therefore, to turn DBROWCOUNT "off", turn it on with a count of 0.

DBSETTIME

Overrides the global DB-Library time-out (set using **dbsettime**) and sets a new DB-Library time-out for a specific DBPROCESS connection. When setting this option, supply a parameter that specifies the connection specific DB-Library time-out in seconds.

DBSHOWPLAN

Generates a description of the processing plan after compilation and continues executing the query.

DBSTAT

Determines, after each query, when performance statistics (CPU time, elapsed time, I/O, and so on) will be returned to the host. DBSTAT takes one of two parameters: *io*, for statistics about SQL Server internal I/O; and *time*, for information about SQL Server's parsing, compilation, and execution times. DB-Library receives these statistics as informational messages, and applications can access them through the user-supplied message handler.

DBSTORPROCID

Sends the stored procedure ID to the host before sending rows generated by the stored procedure.

DBTEXTLIMIT

Causes DB-Library to limit the size of returned **text** or **image** values. When setting this option, supply a parameter with the same length, in bytes, as the longest **text** or **image** value that your program can handle. DB-Library will read but ignore any part of a **text** or **image** value that goes over this limit.

In the case of huge **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**, use the DBTEXTSIZE option instead of DBTEXTLIMIT.

DBTEXTSIZE

Causes SQL Server to limit the size of returned **text** or **image** values. When setting this option, supply a parameter with the same length, in bytes, as the longest **text** or **image** value that SQL Server should return.

Note that, in programs that allow ad hoc queries, the application user can override this option with the Transact-SQL SET TEXTSIZE command. To set a **text** limit that the user cannot override, use the DBTEXTLIMIT option instead.

DBBUFFER, DBNOAUTOFREE, and DBTEXTLIMIT are DB-Library options. That is, they affect DB-Library but are not sent to SQL Server. The others are SQL Server options (options that get sent to the SQL Server). You can set them with Transact-SQL.

As mentioned in the preceding descriptions, certain options take parameters. The following table lists these options and the possible values of their parameters.

Option	Possible parameter values	
DBBUFFER	0 to 32767	
DBOFFSET	select, from, table, order, compute, statement,	
	procedure, execute, or param	
DBROWCOUNT	0 to 2,147,483,647	
DBSTAT	io or time	
DBTEXTLIMIT	0 to 65,534 for 16-bit DB-Library	
	0 to 2,147,483,647 for 32-bit DB-Library	
DBTEXTSIZE	0 to 2,147,483,647	

The function **dbsetopt** requires you to specify parameters when setting any of the options in the preceding table. On the other hand, the functions **dbclropt** and **dbisopt** require you to specify a parameter only for DBOFFSET and DBSTAT, because they can have simultaneous multiple settings, which require further definition before being cleared or checked.

Note that parameters specified in calls to **dbsetopt**, **dbclropt**, and **dbisopt** are always passed as character strings and enclosed in quotation marks, even if they are numeric values.

Using DB-Library for C Data Types

To use Microsoft® SQL Server[™] 2000 data types, include the Sqlfront.h and Sqldb.h header files in your program.

Data Types

The following list describes Microsoft® SQL Server[™] 2000 data types. The **dbconvert** and **dbwillconvert** functions use these types. In addition, **dbcoltype**, **dbalttype**, **dbrettype** each returns one of these types.

Data type	Description	
SQLARRAY	1 byte with variable length	
SQLBINARY	Binary	
SQLBIT	Bit	
SQLCHAR	Char	
SQLDATETIM4	4-byte smalldatetime	
SQLDATETIME	8-byte datetime	
SQLDATETIMN	datetime or smalldatetime with null values	
	allowed	
SQLDECIMAL	Decimal	
SQLFLT4	4-byte real	
SQLFLT8	8-byte float	
SQLFLTN	float or real with null values allowed	
SQLIMAGE	Image	
SQLINT1	1-byte tinyint	
SQLINT2	2-byte smallint	
SQLINT4	4-byte int	
SQLINTN	tinyint, smallint, or int with null values allowed	
SQLMONEY4	4-byte smallmoney	
SQLMONEY	8-byte money	
SQLMONEYN	money or smallmoney with null values allowed	
SQLNUMERIC	Numeric	
SQLTEXT	Text	
SQLVARBINARY	Varbinary	
SQLVARCHAR	Varchar	

Type Definitions

DB-Library functions use the following type definitions. Use these when defining C program variables, particularly those in **dbbind**, **dbaltbind**, **dbconvert**, and **dbdata**.

DBCHAR; // char and text typedef char typedef unsigned char DBBINARY; // binary and image typedef unsigned char DBTINYINT; // 1-byte tinyint DBSMALLINT; // 2-byte smallint typedef short typedef unsigned short DBUSMALLINT; // Unsigned 2-byte integer typedef long DBINT; // 4-byte int DBFLT4; typedef float // 4-byte real // 8-byte float typedef double DBFLT8; typedef unsigned char DBBIT; // bit typedef unsigned char DBBOOL; // Boolean typedef long DBMONEY4; // 4-byte smallmoney

```
typedef struct dbmoney // 8-byte money
{
    DBINT mnyhigh;
    ULONG mnylow;
} DBMONEY;
typedef struct dbdatetime4 // 4-byte smalldatetime
{
    USHORT numdays; // Days since Jan 1, 1900
    USHORT nummins; // Minutes since midnight
} DBDATETIM4;
typedef struct dbdatetime // 8-byte datetime
{
```

```
DBINT dtdays; // Days since Jan 1, 1900
```

ULONG dttime; // 300ths of a second since midnight } DBDATETIME;

```
#define MAXNUMERICLEN 16
```

```
typedef struct dbnumeric // Numeric (and decimal)
{
    BYTE precision; // Precision
    BYTE scale; // Scale
    BYTE sign; // 1 = Positive, 0 = Negative
    BYTE val[MAXNUMERICLEN]; // Padded little-endian value
} DBNUMERIC;
```

typedef DBNUMERIC DBDECIMAL; // Decimal

```
#define DBMAXCHAR 256
```

```
typedef struct dbvarychar // Pascal-type string
{
```

```
DBSMALLINT len; // Character count
DBCHAR str[DBMAXCHAR]; // Nonterminated string
} DBVARYCHAR;
```

```
typedef struct dbvarybin // Pascal-type byte array {
```

DBSMALLINT len; // Byte count

```
BYTE array[DBMAXCHAR]; // Nonterminated array } DBVARYBIN;
```

Error Messages

The table in Errors lists the errors alphabetically with severities. The table in Error Severities summarizes the severity levels. Number values corresponding to the errors are passed to the currently installed, user-supplied error handler. For information about creating an error-handler function for your application, see <u>dberrhandle</u>.

To access these error definitions, include the Sqlfront.h and Sqldb.h header files in your program.

Errors

The following error values are defined in the header file Sqlfront.h. Note that errors with a severity of EXCOMM also have a network-related error message appended to the *dberrstr* value. EXCOMM errors also have network-specific error information in *oserr* and *oserrstr*.

Error			
#	Error	Severity	Description
10000	SQLEMEM	EXRESOURCE	Unable to allocate sufficient
			memory.
10001	SQLENULL	EXPROGRAM	NULL DBPROCESS pointer
			encountered.
10002	SQLENLOG	EXCONSISTENCY	NULL LOGINREC pointer
			encountered.
10003	SQLEPWD	EXUSER	Login incorrect.
10004	SQLECONN	EXCOMM	Unable to connect:
			Microsoft [®] SQL Server TM
			2000 is unavailable or does
			not exist.
10005	SQLEDDNE	EXINFO	DBPROCESS is dead or not
			enabled.
10006	SQLENULLO	EXCONSISTENCY	Attempt to login with NULL
			LOGINREC.
10007	SQLESMSG	EXSERVER	General SQL Server error:
			Check messages from SQL
			Server.
10008	SQLEBTOK	EXCOMM	Bad token from SQL Server:
			Datastream processing out of
			synchronization.
10009	SQLENSPE	EXPROGRAM	General nonspecific DB-
			Library error.
10010	SQLEREAD	EXCOMM	Read from SQL Server failed.
10011	SQLECNOR	EXPROGRAM	Column number out of range.

10012	SQLETSIT	EXINFO	Attempt to call dbtsput with
			an invalid timestamp.
10013	SQLEPARM	EXCONSISTENCY	Invalid parameter in DB-
			Library function reference.
10014	SQLEAUTN	EXPROGRAM	Attempt to update the
			timestamp of a table with no
			timestamp column.
10015	SQLECOFL	EXCONVERSION	Data conversion resulted in
			overflow.
10016	SQLERDCN	EXCONVERSION	Requested data conversion
			does not exist.
10017	SQLEICN	EXPROGRAM	Invalid value for <i>computeid</i> or
			invalid compute column
			number.
10018	SQLECLOS	EXCOMM	Error in closing network
			connection.
10019	SQLENTXT	EXPROGRAM	Attempt to get text
			point/timestamp from a
			non text column.
10020	SQLEDNTI	EXPROGRAM	Attempt to use dbtxtsput to
			put a new text timestamp into
			a column whose data type is
			neither SQLTEXT nor
			SQLIMAGE.
10021	SQLETMTD	EXPROGRAM	Attempt to send too much text
			data through dbmoretext .
10022	SQLEASEC	EXPROGRAM	Attempt to send an empty
			command buffer to the SQL
			Server.
10023	SQLENTLL	EXUSER	Name too long for
			LOGINREC field.
10024	SQLETIME	EXTIME	SQL Server connection timed
			out.
10025	SQLEWRIT	EXCOMM	Write to SQL Server failed.
10026	SQLEMODE	EXCOMM	Network connection not in

			correct mode; invalid SQL
			Server connection.
10027	SQLEOOB	EXCOMM	Error in sending out-of-band data to SQL Server.
10028	SQLEITIM	EXPROGRAM	Illegal time-out value specified.
10029	SQLEDBPS	EXRESOURCE	Maximum number of DBPROCESS structures already allocated.
10030	SQLEIOPT	EXPROGRAM	Attempt to use invalid dboption .
10031	SQLEASNL	EXPROGRAM	Attempt to set fields in a null LOGINREC.
10032	SQLEASUL	EXPROGRAM	Attempt to set unknown LOGINREC field.
10033	SQLENPRM	EXPROGRAM	NULL parameter not allowed for this dboption .
10034	SQLEDBOP	EXPROGRAM	Invalid or out of range <i>dbn</i> parameter.
10035	SQLENSIP	EXPROGRAM	Negative starting index passed to dbstrcpy .
10036	SQLECNULL	EXPROGRAM	NULL destination variable not allowed.
10037	SQLESEOF	EXCOMM	Unexpected EOF from SQL Server.
10038	SQLERPND	EXPROGRAM	Attempt to initiate a new SQL Server operation with results pending.
10039	SQLECSYN	EXCONVERSION	Attempt to convert data stopped by syntax error in source field.
10040	SQLENONET	EXCOMM	DB-Library network communications layer not loaded.
10041	SQLEBTYP	EXPROGRAM	Unknown bind type passed to
			DB-Library function.
--------	-----------	---------------	---
10042	SQLEABNC	EXPROGRAM	Attempt to bind to a
			nonexistent column.
10043	SQLEABMT	EXPROGRAM	User attempted a dbbind with
			mismatched column and
			variable types.
10044	SQLEABNP	EXPROGRAM	Attempt to bind using NULL
100.45			pointers.
10045	SQLEBNCR	EXPROGRAM	Attempt to bind user variable
100.40			to a nonexistent compute row.
10046	SQLEAAMT	EXPROGRAM	User attempted a dbaltbind
			with mismatched column and
10047	SOLENVID	ΕΥΝΟΝΕΛΤΛΙ	The conver did not grant a
10047	SQLEINAID	LANONFAIAL	distributed-transaction ID
100/18	SOI FIFNB	FXPROGRAM	Invalid field number passed to
10040			bcp_control.
10049	SQLEKBCO	EXINFO	1000 rows successfully bulk
			copied to host file.
10050	SQLEBBCI	EXINFO	Batch successfully bulk
			copied to SQL Server.
10051	SQLEKBCI	EXINFO	1000 rows sent to SQL
			Server.
10052	SQLEBCWE	EXNONFATAL	I/O error while writing bcp
			data file.
10053	SQLEBCNN	EXUSER	Attempt to bulk copy a null
			value into a server column
			that does not accept null
10054		EVCONSISTENCY	Attempt to bulk conv an
10054	SQLEDCOR	EACONSISTENCY	Attempt to burk copy an oversized row to SOL Server
10055	SOI FRODI	FXDROCPAM	Call bcn init before any
10033			other bcp_nint before any
10056	SOLEBCPN	EXPROGRAM	Use bcn_bind_bcn_collen
10000			and bcp_colptr only after
			I I I I I I

			calling bcp_init with the copy
			direction set to DB_IN.
10057	SQLEBCPB	EXPROGRAM	Do not use bcp_bind after
			bcp_init has been passed a
			nonnull data file name.
10058	SQLEVDPT	EXUSER	For bulk copy, all variable-
			length data must have either a
			length prefix or a terminator
			specified.
10059	SQLEBIVI	EXPROGRAM	Use bcp_columns and
			<pre>bcp_colfmt only after</pre>
			bcp_init has been passed a
			valid data file.
10060	SQLEBCBC	EXPROGRAM	Call bcp_columns before
			bcp_colfmt.
10061	SQLEBCFO	EXUSER	Host files must contain at
			least one column: bcp .
10062	SQLEBCVH	EXPROGRAM	Call bcp_exec only after
			bcp_init has been passed a
			valid host file.
10063	SQLEBCUO	EXRESOURCE	Unable to open host datafile:
			bcp.
10064	SQLEBUOE	EXRESOURCE	Unable to open error file:
			bcp.
10065	SQLEBWEF	EXNONFATAL	I/O error while writing bcp
			error file.
10066	SQLEBTMT	EXPROGRAM	Attempt to send too much text
			data with bcp_moretext .
10067	SQLEBEOF	EXNONFATAL	Unexpected EOF encountered
			in bcp datafile.
10068	SQLEBCSI	EXCONSISTENCY	Host-file columns may be
			skipped only when copying
			into the server.
10069	SQLEPNUL	EXCONSISTENCY	NULL program pointer
			encountered.

10070	SQLEBSKERR	EXCONSISTENCY	Cannot seek in data file.
10071	SQLEBDIO	EXPROGRAM	Bad bulk-copy direction.
10072	SQLEBCNT	EXUSER	Attempt to use bulk copy with a nonexistent server table.
10073	SQLEMDBP	EXPROGRAM	Attempt to set maximum number of DPPROCESS structures lower than 1.
10075	SQLCRSINV	EXPROGRAM	Invalid cursor statement.
10076	SQLCRSCMD	EXPROGRAM	Attempt to call cursor functions when there are commands waiting to be executed.
10077	SQLCRSNOIND	EXINFO	One of the tables involved in the cursor statement does not have a unique index.
10078	SQLCRSDIS	EXPROGRAM	Cursor statement contains one of the disallowed phrases COMPUTE, UNION, FOR BROWSE, or SELECT INTO.
10079	SQLCRSAGR	EXPROGRAM	Aggregate functions are not allowed in a cursor statement.
10080	SQLCRSORD	EXPROGRAM	Only fully keyset driven cursors can have ORDER BY, GROUP BY, or HAVING PHRASES clauses.
10081	SQLCRSMEM	EXPROGRAM	Keyset or window scroll size exceeds the memory limitations of this computer.
10082	SQLCRSBSKEY	EXPROGRAM	Keyset cannot be scrolled backward in mixed cursors with a previous fetch type.
10083	SQLCRSNORES	EXINFO	Cursor statement generated no results.
10084	SQLCRSVIEW	EXPROGRAM	A view cannot be joined with

			another table or a view in a
			cursor statement.
10085	SQLCRSBUFR	EXPROGRAM	Row buffering should not be turned on when using cursor functions.
10086	SQLCRSFROWN	EXINFO	Row number to be fetched is outside valid range.
10087	SQLCRSBROL	EXPROGRAM	Backward scrolling cannot be used in a forward scrolling cursor.
10088	SQLCRSFRAND	EXPROGRAM	Fetch types RANDOM and RELATIVE can only be used within the keyset of keyset driven cursors.
10089	SQLCRSFLAST	EXPROGRAM	Fetch type LAST requires fully keyset driven cursors.
10090	SQLCRSRO	EXPROGRAM	Data locking or modifications cannot be made in a READONLY cursor.
10091	SQLCRSTAB	EXPROGRAM	Table name must be determined in operations involving data locking or modifications.
10092	SQLCRSUPDTAB	EXPROGRAM	Update or insert operations using bind variables require single table cursors.
10093	SQLCRSUPDNB	EXPROGRAM	Update or insert operations cannot use bind variables when binding type is NOBIND.
10094	SQLCRSVIIND	EXPROGRAM	The view used in the cursor statement does not include all the unique index columns of the underlying tables.
10095	SQLCRSNOUPD	EXINFO	Update or delete operation did not affect any rows.

10096	SQLCRSOS2	EXPROGRAM	Cursors are not supported for this server.
10097	SQLEBCSA	EXPROGRAM	The BCP hostfile %s contains only %ld rows. Skipping all of these rows is not allowed.
10098	SQLCRSRO	EXPROGRAM	Data locking or modifications cannot be made in a READONLY cursor.
10099	SQLEBCNE	EXPROGRAM	The table %s contains only %ld rows. Copying up to row %ld is not possible.
10100	SQLEBCSK	EXPROGRAM	The table %s contains only % <i>ld</i> rows. Skipping all of these rows is not allowed.
10101	SQLEUVBF	EXPROGRAM	Attempt to read unknown version of bcp format file.
10102	SQLEBIHC	EXPROGRAM	Incorrect host-column number found in bcp format file.
10103	SQLEBWFF	EXRESOURCE	I/O error while reading bcp format file.
10104	SQLNUMVAL	EXPROGRAM	The data stored in the DBNUMERIC/DBDECIMAL structure is invalid.
10105	SQLEOLDVR	EXPROGRAM	The SQL Server's TDS is obsolete with this version of DB-Library.
10106	SQLEBCPS	EXPROGRAM	The row length exceeds SQL Server's maximum allowable size.
10107	SQLEDTC	EXRESOURCE	Microsoft Distributed Transaction Coordinator call failed.
10108	SQLENOTIMPL	EXPROGRAM	This function is not supported on this platform at this time.
10109	SQLENONFLOAT	EXPROGRAM	float conversion attempt

			failed. The source is invalid.
10110	SQLECONNFB	EXCOMM	Unable to connect: DB Server
			is unavailable or does not
			exist - will attempt a fallback
			connection.

Error Severities

Error severities have numerical equivalents. When an error occurs or when a message is sent, these numerical equivalents are passed to the currently installed, user-supplied error handler.

Error severity levels are defined in the Sqlfront.h header file. Your program must include Sqlfront.h if it refers to these severity levels.

Error severity	Severity number	Description
EXINFO	1	Informational, nonerror.
EXUSER	2	User error.
EXNONFATAL	3	Nonfatal error.
EXCONVERSION	4	Error in DB-Library data conversion.
5		The server has returned an error flag.
EXTIME	6	Time-out period exceeded while waiting for a response from the server; the DBPROCESS is still alive.
EXPROGRAM	7	Coding error in user program.
EXRESOURCE	8	Running out of resources; the DBPROCESS may be dead.
EXCOMM	9	Failure in communication with server; the DBPROCESS is dead.
EXFATAL	10	Fatal error; the DBPROCESS is dead.
EXCONSISTENCY	11	Internal software error; notify your primary support provider.

DB-Library for C Samples

The DB-Library for C sample program files are stored as projects in their own directories. All samples are designed for use with Mixed Mode security. Comments explaining how to implement Windows Authentication mode are provided in each sample.

When the Microsoft® SQL Server[™] sample files are installed, the sample sources are installed to subdirectories of:

C:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Samples\Dblib

Example8 - Handling Procedure Output Parameters

Example8 illustrates DB Library support for output parameters in Microsoft® SQL Server[™] stored procedures.

Default Location

C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\DBLib\Example8

Running the Sample

- 1. Open Example8.dsw in Microsoft Visual C++® 6.0.
- 2. From the **Tools** menu, choose **Options**, and then click the **Directories** tab.
- 3. From the **Show directories for** box, choose Include files and Library files, and ensure that the following directories (as appropriate) are included:
 - Include files: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Include
 - Library files: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Lib
- 4. Ensure that the hard-coded server name, user name, and password are correct.
- 5. Compile the program.
- 6. Create the stored procedure, rpctest by running C:\Program Files\Microsoft SQL

Server\80\Tools\Devtools\Samples\DBLib\Example8\Createsp.sql).

- 7. Open a Command Prompt window, and then change the current directory to C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\DBLib\Example8.
- 8. Run the program by entering "example8".

Functions Illustrated

Dbinit	dbresults
Dbmsghandle	dbprrow
Dberrhandle	dbnumrets
Dblogin	dbretname
DBSETLUSER	dbrettype
DBSETLPWD	dbprtype
DBSETLAPP	dbretlen
DBSETLVERSION	dbretdata
Dbopen	dbhasretstat
Dbrpcinit	dbretstatus
Dbrpcparam	dbexit
Dbrpcsend	DBDEAD
Dbsqlok	

See Also

<u>Samples</u>

SQLExamp: Determining Column Widths

SQLExamp illustrates handling of various data types for consistent output. The example processes a user's Transact-SQL statement and formats result rowset data for display.

Default Location

C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\DBLib\SQLExamp

Functions Illustrated

Dbinit	dbnextrow
Dbsettime	dbnumcols
Dbmsghandle	dbcoltype
Dberrhandle	dbcolinfo
Dblogin	dbcollen
DBSETLUSER	dbcolname
DBSETLHOST	dbdata
DBSETLVERSION	dbdatlen
Dbopen	dbconvert
Dbcmd	dbcancel
Dbsqlexec	DBCOUNT
Dbresults	dbclose

See Also

SQLTestC - SELECT Statement Processing

SQLTestc is a console mode version of the SQL Test example. The example issues a simple SELECT statement against the **pubs** database and prints the returned result set rows to the console.

Default Location

C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\DBLib\SQLTestC

- 1. Open the Sqltestc.dsw in Microsoft® Visual C++® 6.0.
- 2. From the **Tools** menu, choose **Options**, and then click the **Directories** tab.
- 3. From the **Show directories for** box, choose Include files and Library files, and ensure that the following directories (as appropriate) are included:
 - Include files: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Include
 - Library files: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Lib
- 4. Ensure that the hard-coded server name, user name, and password are correct.
- 5. Compile the program.
- 6. Open a Command Prompt window, and then change the current directory to C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\DBLib\Sqltestc.

7. Run the program by entering "sqltestc".

Functions Illustrated

Dbinit	dbuse
Dbmsghandle	dbcmd
Dberrhandle	dbsqlexec
Dblogin	dbresults
DBSETLUSER	dbbind
DBSETLPWD	dbnextrow
DBSETLAPP	dbexit
DBSETLVERSION	DBDEAD
Dbopen	

See Also

SQLTestN

SQLTestN is a Win32® API, graphical user interface sample. The sample illustrates connecting to a Microsoft® SQL Server[™] and executing a simple SELECT statement.

Default Location

C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\DBLib\SQLTestN

- 1. Open the sqltestn.dsw in Microsoft Visual C++® 6.0.
- 2. From the **Tools** menu, choose **Options**, and then click the **Directories** tab.
- 3. From the **Show directories for** box, choose Include files and Library files, and ensure that the following directories (as appropriate) are included:
 - Include files: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Include
 - Library files: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Lib
- 4. Ensure that the hard-coded user name is correct. The password is assumed to be NULL.
- 5. Compile the program.
- 6. Open a Command Prompt window, and then change the current directory to C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\DBLib\Sqltestn.

7. Run the program by entering "sqltestn".

Functions Illustrated

Dbbind	dbmsghandle
Dbclose	dbnextrow
Dbcmd	dbopen
Dberrhandle	dbresults
Dbexit	DBSETLUSER
Dbfreelogin	DBSETLVERSION
Dbinit	dbsqlexec
DBLOCKLIB	DBUNLOCKLIB
Dblogin	dbuse

See Also

TextCopy - Handling text and image data

TextCopy is a Microsoft Foundation Class (MFC) C++ example that illustrates handling Microsoft® SQL Server[™] **text** and **image** data types. The example moves data either to or from a single server column and row, and a user-specified operating system file.

Default Location

C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\DBLib\TextCopy

Functions Illustrated

Dbinit	dbuse
Dbmsghandle	dbcmd
Dberrhandle	dbnumcols
Dbsettime	dbcoltype
Dbsetlogintime	dbreadtext
Dblogin	dbnextrow
DBSETLUSER	dbtxtptr
DBSETLPWD	dbtxtimestamp
DBSETLAPP	dbwritetext
DBSETLHOST	dbsqlok
Dbopen	dbmoretext
Dbfreelogin	dbcancel
Dbsetopt	dbclose
Dbsqlexec	DBDEAD
Dbresults	

See Also