Rocket UniData

Using the UniBasic SQL Client Interface (BCI)

Version 8.2.1

July 2017
UDT-821-BCI-1

Notices

Edition

Publication date: July 2017
Book number: UDT-821-BCI-1
Product version: Version 8.2.1

Copyright

Trademarks

Examples

This information might contain examples of data and reports. The examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

License agreement

Note: This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Rocket Global Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

Country	Toll-free telephone number
United States	1-855-577-4323
Australia	1-800-823-405
Belgium	0800-266-65
Canada	1-855-577-4323
China	400-120-9242
France	08-05-08-05-62
Germany	0800-180-0882
Italy	800-878-295
Japan	0800-170-5464
Netherlands	0-800-022-2961
New Zealand	0800-003210
South Africa	0-800-980-818
United Kingdom	0800-520-0439

Contacting Technical Support

The Rocket Community is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Community and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Community or to request a Rocket Community account, go to www.rocketsoftware.com/support.

In addition to using the Rocket Community to obtain support, you can use one of the telephone numbers that are listed above or send an email to support@rocketsoftware.com.

Contents

Chapter 1: Introduction

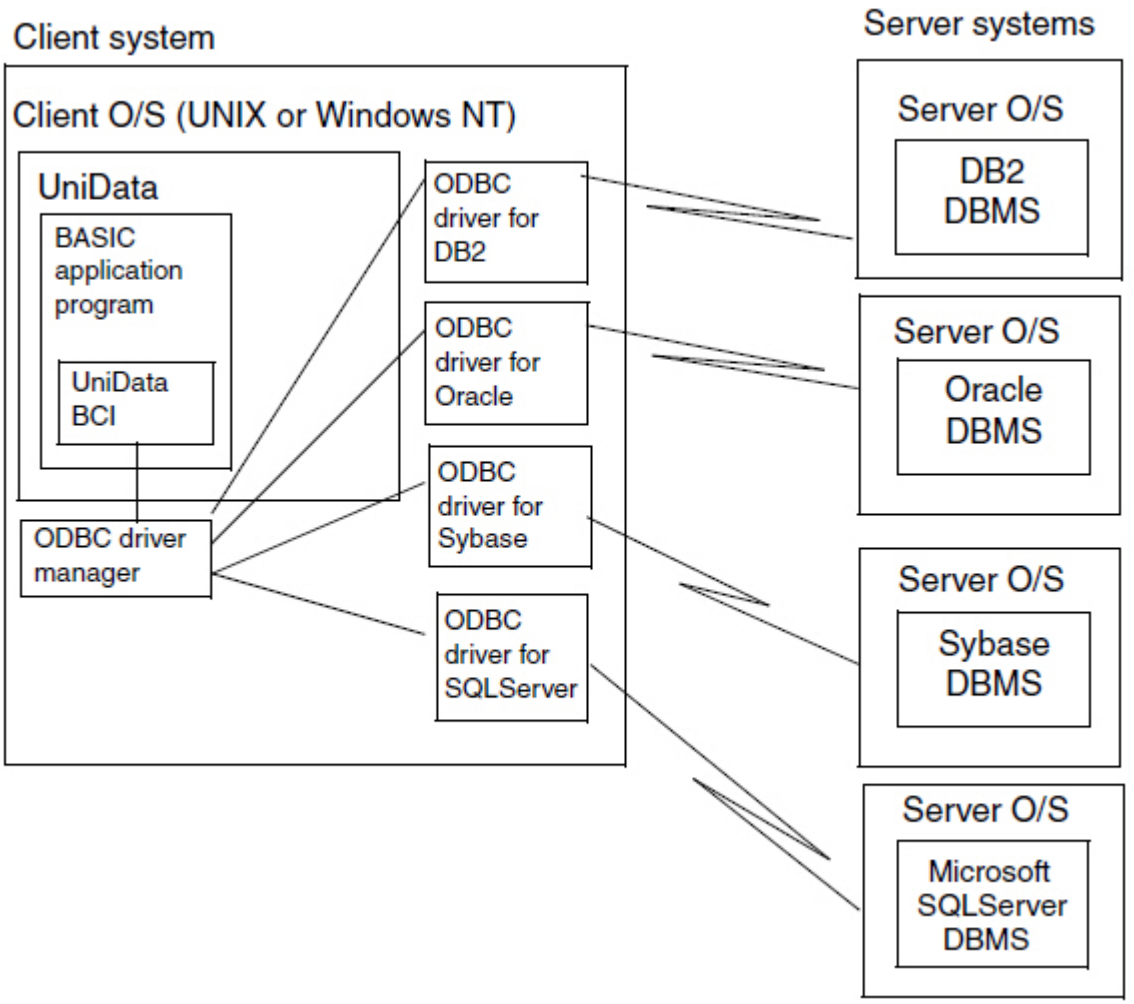
The UniBasic SQL Client Interface (Rocket UniData BCI) is an application programming interface (API) that makes UniData a client in a client/server environment. The server should be an ODBC data source running a relational DBMS such as DB2, Oracle, Microsoft SQLServer, or Sybase.

You use UniData BCI to connect to one or more data sources.

ODBC data sources

To connect to an ODBC data source, an ODBC driver manager and suitable ODBC drivers for the data sources you want to connect to must be installed on the client system.

Once connected to any data source, UniData BCI lets you read data from and write data to the data source. The following illustration shows how your application program can access the capabilities of the server DBMS.



UniData BCI also includes the ECL `CONNECT` command, which lets users access data sources interactively. For information about the `CONNECT` command, see Using the CONNECT command, on page 12.

UniData BCI is based on the core-level definition of the Microsoft Open Database Connectivity (ODBC) interface. The ODBC interface lets you write programs that can operate across a wide range of data sources. For complete information about the ODBC interface, see *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Additional UniBasic functions

UniBasic includes a set of functions that make up the UniData BCI Interface. A client application program uses these functions to do the following:

- Allocate resources for connections
- Connect to one or more remote data sources
- Send SQL statements to the data source for execution
- Call procedures stored on the data source for execution
- Receive results row by row from SELECT statements
- Insert, update, and delete rows using SQL data manipulation statements
- Create and drop tables and views using SQL data definition statements
- Receive status and error information from the data source
- Disconnect from the data source

The CONNECT command

UniData BCI provides a utility, invoked with the `CONNECT` command, that lets you connect to a server DBMS and interactively manipulate and display data from that system on the client system.

For information about the `CONNECT` command, see Using the CONNECT command, on page 12.

System requirements

To use UniData BCI to access an ODBC database, you need the following:

- TCP/IP hardware and software installed on both client and server systems
- At least one DBMS installed on a server system
- ODBC driver manager and ODBC driver for the data source, installed on the client system
- Release 5.2 or later of UniData installed on the client system

Setting up the ODBC environment on UniData for UNIX

You must perform the following tasks before using UniData BCI on UniData for UNIX.

Install the ODBC environment

Install the ODBC environment according to the vendor's instructions. The ODBC installation normally consists of the following steps:

1. Install the ODBC driver manager.

2. Install the ODBC driver and client components, if necessary. For example, on Oracle, you must install the ODBC driver and the client component.
3. Set the environment variable for your ODBC shared libraries. The name of this environment variable is not the same on all UNIX systems.

   For example, the environment variable is called LD_LIBRARY_PATH on Solaris systems, SHLIB_PATH on HP systems, and so forth. If this environment variable is not properly set, running UniData BCI programs may produce errors similar to the following example:

   ```
   ld.so.1: uvsh: fatal: libxxxx: can't open file: errno=2
   ```

   xxxx may be some unrecognizable combination of letters and numbers. To correct this problem, set your environment according to the vendor's instructions.
4. Set the ODBCINI environment variable to the location of the file containing the names and descriptions of the ODBC data sources.

Determine location of ODBC shared libraries

Determine where the ODBC shared library `libodbc.xx` resides. For example, the default location of the Merant driver is `/opt/odbc/lib`.

Relink ODBC shared libraries

On UNIX systems, UniData installs a dummy ODBC shared library in the `/udthome/lib/uddlls` directory during the installation process.

This library has the name `libodbc.`*xx*, where *xx* is supplied by the system on which you are running. The installation program creates a symbolic link from `/.udlibs` to the `/udthome/lib/uddlls` directory. The udt, udsrvd, and udapi_slave modules look for their shared libraries in this directory, so it is necessary that this symbolic link not be broken.

You must execute the `relink.udlibs` script for any of the following reasons:

- To initially link to the actual ODBC driver libraries that you are using on your client machine (where UniData is running).
- If for any reason the symbolic link is broken.
- If the system administrator moves the shared libraries to another directory.

The `relink.udlibs` script is located in /udthome/bin. To relink the shared libraries, use the following syntax:

```
relink.udlibs pathname
```

pathname is the full path of the directory containing the shared libraries. For example, to relink to the Merant driver library, enter the following command:

```
% relink.udlibs /opt/odbc/lib
```

Note: You do not need to stop and start UniData to use the new shared libraries. However, users currently running a UniData session will continue to use the old shared libraries until they exit and reenter their current UniData session.

Setting up the ODBC environment on UniData for Windows platforms

On Windows platforms, you must have ODBC and the appropriate drivers installed according to the vendor's instructions. The Microsoft ODBC data source administrator is then used to define ODBC data sources prior to using them in UniData BCI.

Chapter 2: Getting started

This chapter describes how to run the UniData BCI demonstration program.

Using UniData BCI

After you set up the ODBC environment, you can perform any of the following tasks:

- Run the UniData BCI demonstration program BCI.DEMO, located in /*udthome*/demo/BP on UniData for UNIX or *udthome*\demo\BP on UniData for Windows Platforms.
- Use the CONNECT command to connect to a data source
- Use UniBasic to write a UniData BCI application program

See Using the CONNECT command, on page 12 for details about the CONNECT command. See Using UniData BCI, on page 22 and UniData BCI functions, on page 31 for details about UniData BCI. The next section describes how to run the demonstration program.

Running the demonstration program

When you install UniData, the BCI.DEMO demonstration program is copied into the BP file of the UniData demo account.

UniData BCI demo program, on page 82 contains the source code for BCI.DEMO and explains what it does.

You must specify the datasource, user name, and password for your environment, as shown in the following example:

```
.
.
.
* datasource == Name of the established ODBC datasource
* username == Name of user as defined on the SQL Database Engine
* passwd == Password of user
* ========================================================================
*
datasource = "Server7"
username = "sa"
passwd = ""
RETURN
*
.
.
.
```

Before you run the program, you must compile it, as shown in the following example:

```
:BASIC BP BCI.DEMO -i

Compiling Unibasic: BP/BCI.DEMO in mode 'u'.
compilation finished
```

Note: Make sure you compile the BCI.DEMO program using the -i option with the `BASIC` command. This option makes UniBasic reserved words case-insensitive.

Run the program

To run the program, enter the following at the ECL prompt:

```
:RUN BP BCI.DEMO
```

After the data source accepts your login parameters, the program displays output similar to the following example:

```
Attempting connect to Server7 with user id sa
Deleting local EMPFILE file
Deleting file D_EMPFILE.
Deleting file EMPFILE.
Create file D_EMPFILE, modulo/1,blocksize/1024
Hash type = 0
Create file EMPFILE, modulo/3,blocksize/1024
Hash type = 0
Added "@ID", the default record for UniData to DICT EMPFILE.
EMPFILE is cleared.
Dropping EMPTABLE table in Server7
Creating EMPTABLE table in Server7
Loading row 1 from EMPFILE
Loading row 2 from EMPFILE
Loading row 3 from EMPFILE
Loading row 4 from EMPFILE
Loading row 5 from EMPFILE
ID      NAME            GRADE   CITY
----------------------------------------
E3      Carmen      13      Vienna
E1      Alice           12      Deale
E4      Don                 12      Deale
E2      Betty           10      Vienna
E5      Ed              13      Akron
Exiting bcidemo
:
```

Chapter 3: Using the CONNECT command

This chapter describes how to use the CONNECT command to connect to a data source from a UniData client. You enter the CONNECT command at the ECL prompt. The CONNECT command enables you to submit SQL statements to the data source and receive results at your terminal.

While you are connected to a data source, you can enter any SQL statement understood by the DBMS engine on the data source, including SELECT, INSERT, UPDATE, DELETE, GRANT, and CREATE TABLE. ODBC data sources can use SQL language that is consistent with the ODBC grammar specification as documented in the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

The CONNECT command runs in autocommit mode: that is, all changes made to the data source DBMS are committed immediately. Do not use transaction control statements such as TRANSACTION START, TRANSACTION COMMIT, and TRANSACTION ABORT when you are using CONNECT. For information about transactions, see Transaction management, on page 28.

Command syntax

The syntax of the CONNECT command is as follows:

CONNECT *data.source* [*option setting* [*option setting* …]]

data.source is the name of the data source to which you want to connect.

The data source is an ODBC data source defined on your system. For example, on Windows platforms, a data source is defined in the ODBC Data Source Administrator.

option is any of the following:

- BLOCK
- NULL
- PREFIX
- UDOUT
- VERBOSE
- WIDTH

Note: You can enter an SQL statement on several lines. If a statement line does not end with a semicolon or a question mark and you press Enter, the SQL continuation prompt (SQL+) appears.

Command options

You can specify any option with the CONNECT command. You must specify a setting for the option. The following section describes the options and their possible settings in detail.

BLOCK option

The BLOCK option defines how UniData BCI terminates input statements.

setting is one of the following:

Setting	Description
ON	Enables block mode. In this mode you can enter a series of SQL statements, ending each with a ; (semicolon). To terminate the block of SQL statements, press Enter immediately after an SQL+ prompt.
OFF	Disables block mode. In this mode if you type a semicolon at the end of a line of input, UniData BCIthe SQL Client Interface terminates your input and sends it to the data source. This is the default setting.
string	Enables block mode (see ON, above). *string* must be from 1 to 4 characters. To terminate the block of SQL statements, enter *string* immediately after an SQL+ prompt.

For more details, see Using block mode, on page 16.

NULL option

The way UniData BCI treats null values coming from the data source depends on the setting of the NULL_FLAG parameter in the `udtconfig` file.

NULL FLAG	Description
0	Remote nulls are translated to or from the data source as an empty string.
1	Remote nulls are translated to or from the data source as the null value mark.

The NULL option defines how to display the SQL null value. This option is only valid if the NULL_FLAG is set to 1 in the `udtconfig` file, located in `/usr/ud73/include`. *setting* is one of the following:

Setting	Description
SPACE	Displays the SQL null value as a blank space.
NOCONV	Displays the SQL null value as defined by null value mark setting in UDTLANGCONFIG.
string	Displays the SQL null value as *string*. The string can be from 1 to 4 characters. By default, null is displayed as the four-character string NULL.

PREFIX option

The PREFIX option defines the prefix character for local commands. *setting* is any valid prefix character. The default prefix character is a period (.). You can use only the following characters as the prefix character:

Character	Description
!	Exclamation point.
@	At sign.
#	Hash sign.
$	Dollar sign.
%	Percent.
&	Ampersand.
*	Asterisk.
/	Slash.
\	Backslash.
:	Colon.

Character	Description
=	Equal sign.
+	Plus sign.
-	Minus sign.
?	Question mark.
(	Left parenthesis.
)	Right parenthesis.
{	Left brace.
}	Right brace.
[	Left bracket.
]	Right bracket.
‘	Left quotation mark.
‘	Right quotation mark.
.	Period.
|	Vertical bar.
"	Double quotation mark.
,	Comma.

For more details, see Using local commands, on page 16.

UDOUT option

The UDOUT option specifies how to handle output from `SELECT` statements executed on the data source.

setting is either:

Setting	Description
filename	Stores output in *filename* on the client, then displays the output from *filename*. If the file does not exist, the `CONNECT` command creates it.
OFF	Displays output from the data source directly on the screen of the client. This is the default setting.

For more details, see Displaying and storing output, on page 17.

VERBOSE option

The VERBOSE option displays extended column information and system messages.

setting is either:

Setting	Description
ON	Enables verbose mode. In this mode the name, SQL data type, precision, scale, and display size are displayed for each column definition when selecting data from the data source. Error messages are displayed in extended format that includes the type of call issued, status, SQLSTATE, error code generated by the data source, and the complete error text.
OFF	Disables verbose mode. This is the default setting.

WIDTH option

The WIDTH option defines the width of display columns. *setting* is one of the following:

Setting	Description
col#,*width*	Sets the width of column *col#* to *width*. Do not enter a space after the comma. Specify *col#* as * (asterisk) to set the width of all columns. *width* can be from 4 to the maximum line length allowed by your terminal. The default width for all columns is 10.
T	Truncates data that is wider than the *width* you specify. This is the default setting.
F	Folds data that is wider than the specified *width* onto multiple lines.
?	Displays the current column width settings, and tells whether data will be truncated or folded.

Logging on to the data source

After you enter the `CONNECT` command and the initial validity checks succeed, UniData BCI prompts to enter your name and password to connect to the data source.

The user name and password must be valid on the server. The default user name is your logon name on the client system.

After you log on successfully to the server operating system, and if the DBMS is currently running, UniData BCI prompts you to enter the logon parameters you use to access the DBMS on the server.

After accepting the DBMS login parameters, the data source prompt appears, as shown in the following example:

`data.source.name``>`

The next sections show examples of the logon sequence on different systems. Brackets enclose default entries, which you can accept by pressing Enter.

Logging in to an ODBC data source

The following example shows what happens when you use `CONNECT` to log in to an ODBC data source:

```
>CONNECT odbc-ora
Enter username for connecting to 'odbc-ora' DBMS [VEGA\george]: fred
Enter password for fred:
odbc-ora>
```

Errors when logging on to a data source

UniData BCI performs several validity checks when you enter `CONNECT` *`data.source`*. It runs these checks before prompting you for your user name and password.

The most common errors that can occur at this point are the following:

- On Windows platforms, the data source name is incorrect or undefined in ODBC Data Source Administration.

- The server does not respond. This can be due to problems on the network or problems with the server software. You will see something like the following:

```
:CONNECT ENGINEERING7
Enter username for connecting to 'ENGINEERING7' DBMS [claireg]: sa
Enter password for sa:
SQLConnect error: Status = -1 SQLState = 08003 Natcode = 0
[ODBC] [MERANT][ODBC lib] Connection not open
```

- If the DBMS is not currently running on the server, you will see something like the following:

```
SQLConnect error: Status = -1 SQLState = S1000 Natcode = 9352

[ODBC] [INTERSOLV][ODBC Oracle driver][Oracle]ORA-09352: Windows 32-bit

Two-Task driver unable to spawn new ORACLE task
```

The failure of a connection to an ODBC data source generates error messages particular to that ODBC driver and database server.

Exiting from the data source

To exit from the data source, enter `.q`.

Using block mode

Block mode lets you send a series of SQL statements, each terminated with a semicolon, to the data source as a single block. For instance, you would use block mode to send PL/SQL blocks or stored procedures to an ODBC data source running Oracle.

If block mode is enabled, a semicolon does not terminate your input. SQL statements are sent to the data source for execution only when you press Enter or enter a termination string immediately after an SQL+ prompt. SQL statements ending with a question mark are not sent to the data source; they are stored on the client.

Use the BLOCK option of the `CONNECT` command or the local command `.BLOCK` to enable and disable block mode.

You can enable block mode in two ways. With block mode set to ON, you terminate input by pressing Enter immediately after an SQL+ prompt.

With block mode set to a character string, you enter the character string immediately after an SQL+ prompt to terminate input. For example, you might want to terminate your input with a line such as GO. The string can be up to four characters long. The string is not case-sensitive, so if you specify GO with the BLOCK option or the .B command, for example, you can terminate input with `GO` or `go` .

Using local commands

Commands starting with the designated prefix character are treated as local commands—that is, the client machine processes them. The default prefix character is a . (period). You can change the prefix character by using the PREFIX option of the `CONNECT` command.

Local commands cannot end with a semicolon or a question mark. The following are valid local commands:

Valid local commands	Description
.A *string*	Appends *string* to the most recent SQL statement.
.B [LOCK] *setting*	Enables or disables block mode. *setting* can be ON, OFF, or a character string. For more details, see Using block mode, on page 16.
.C/*old*/*new* [/ [G]]	Changes the first instance of *old* to *new* in the most recent SQL statement. If you use the G (global) option, .C changes all instances of *old* to *new*. You can replace the slash with any valid delimiter character. Valid delimiters are the same as valid prefix characters. For a list of valid delimiters, see the PREFIX option described earlier.
.EXECUTE	Executes the most recent SQL statement.
.M [VDISPLAY] *setting*	Defines how to display value marks in multivalued data. *setting* can be SPACE, NOCONV, or a character.
.N [ULL] *setting*	Defines how to display the SQL null value. *setting* can be SPACE, NOCONV, or a character string.
.P [RINT]	If the most recent SQL statement is SELECT, executes the statement and sends output to logical print channel 0. If the most recent SQL statement is not SELECT, executes the statement.
.Q [UIT]	Exits from CONNECT and returns to the ECL prompt.
.R [ECALL] [*name*]	Displays, but does not execute, the SQL statement stored as *name* in the VOC. If you do not specify *name*, .RECALL displays the most recent SQL statement.
.S [AVE] *name*	Saves the most recent SQL statement as the sentence *name* in the VOC file.
.T [OP]	Clears the screen.
.U [DOUT] *setting*	Specifies how to handle output from SELECT statements. *setting* can be OFF or the name of a file on the client system. For more detail, see Displaying and storing output, on page 17.
.V [ERBOSE] *setting*	Enables or disables verbose mode. *setting* can be ON or OFF. For more details see the VERBOSE option described earlier.
.W [IDTH] *setting*	Defines the width of display columns. For information about how to set and display column widths, see the WIDTH option of the CONNECT command.
.X	Executes the most recent SQL statement.

Displaying and storing output

Use the UDOUT option or the .UDOUT local command to turn UniData output mode on and off. By default, UniData output mode is off, and CONNECT displays output from SQL statements on the screen.

If a row of output is wider than the line length defined for your screen, CONNECT displays each row in the UniData vertical mode, with each column on a separate line. Otherwise, blank columns two characters wide separate display columns. CONNECT folds column headings into several lines if they do not fit on one line, and truncates or folds data that is wider than a column, depending on the WIDTH setting (T or F). An * (asterisk) appears next to truncated data, a - (hyphen) appears next to folded data.

In UniData output mode, CONNECT writes each row of data to a UniData file on the client (the file is first cleared). It then generates a UniData SQL SELECT statement that displays the data in the file.

In both output modes, data is left-justified if its type is CHAR or DATE, text-justified if its type is VARCHAR, and right-justified if it is any other data type.

You can use UniData output mode to transfer data from the data source to your UniData database. This is because output from a SELECT statement is stored in a UniData file. However, each SELECT clears the UniData output file before writing output to it. If you want to save the results of several SELECTs in your UniData database, you must use several UniData output files.

Examples

The following example shows a normal login to an ODBC data source running Sybase. Some data is selected from a table.

```
>CONNECT syb
Enter username for connecting to 'syb' DBMS [hatch]:<Return>Enter password for hatch:
syb>select * from
tedtab1;
                pk  colchar8      colint     colreal
----------  --------  ----------  ----------
         1              New York        9876            3.40000009*
         2              Chicago     543         23.3999996*
2 rows selected
```

Using verbose mode

The next example turns on verbose mode and executes the previous SELECT statement:

```
syb> .v on
syb>.xThere are 4 columns
Column 1 name is:  pk
Column 1 type is:  4  (SQL.INTEGER)
Column 1 prec is:  10
Column 1 scale is: 0
Column 1 dispsize is: 11
Column 2 name is:  colchar8
Column 2 type is:  1  (SQL.CHAR)
Column 2 prec is:  8
Column 2 scale is: 0
Column 2 dispsize is: 8
Column 3 name is:  colint
Column 3 type is:  4  (SQL.INTEGER)
Column 3 prec is:  10
Column 3 scale is: 0
Column 3 dispsize is: 11
Column 4 name is:  colfloat
Column 4 type is:  8  (SQL.DOUBLE)
Column 4 prec is:  15
Column 4 scale is: 0
Column 4 dispsize is: 22
        pk          colchar8      colint    colfloat
----------  --------  ----------  ----------
         1          New York        9876         3.40000009*
         2          Chicago          543        23.3999996*

2 rows selected
syb>.v offsyb>.w fsyb>.x
        pk          colchar8      colint    colfloat
----------  --------  ----------  ----------
        1           New York        9876            3.40000009-
                                                                5
```

```
        2           Chicago          543            23.3999996-
                                                                19

2 rows selected
syb>.w 4,15syb>.x
         pk         colchar8      colint      colfloat
----------  --------  ----------    ----------
          1          New York        9876        3.400000095
          2          Chicago          543            23.399999619

2 rows selected
syb>.wTruncate/Fold mode is:            F
Column width settings are:
                  Column 4:  15
           All other columns: 10
```

Changing the display width of columns

The following example shows what happens when you change the display width to fold data that does not fit (as shown in the previous example, in the colreal column):

```
syb> .w f
syb>.x
         pk             colchar8      colint     colreal
----------  --------  ----------  ----------
          1             New York        9876        3.40000009-
                                                              5
          2             Chicago          543        23.3999996-
                                                               19
2 rows selected
```

By changing the display width of column 4 to 15 characters, you get the following display:

```
syb> .w 4,15
syb>.x
        pk             colchar8      colint          colreal
----------  --------  ----------  ---------------
          1             New York        9876         3.400000095
          2             Chicago          543        23.399999619
2 rows selected
```

To display the current display width settings, use a question mark after the `.W` command, as shown in the following example:

```
syb> .w ?
Truncate/Fold mode is:              F
Column width settings are:
                                Column 4:  15
                All other columns: 10
```

Exiting CONNECT

The next example inserts values into a table, selects and displays them, then quits from `CONNECT`:

```
syb> insert into tedtab3 values (3,9,1,8);
```

```
1 row affected
syb>select * from tedtab3;
        pk              coltinyint      colbit          colsmint
----------  ----------  ----------  ---------------
         1                     255               0                    -32768
         2                           0                     0                     32768
         3                           9                     1                     8

3 rows selected
syb>.q
Disconnecting from 'syb' database
>
```

Using UniData output mode

The following example shows how to use two UniData output files to save the results of two SELECT statements.

First, enter UniData output mode while you are connected to the data source *ora*:

```
ora> .U UDFILE1
Opening file UDFILE1
```

Next, select data from a table in *ora*:

```
ora> SELECT * FROM TEDTAB2;
COLC......   COLD......

detroit             lions
pittsburgh      steelers
new york        giants
3 records listed.
```

Now switch to a different UniData output file and enter another SELECT statement:

```
ora> .U UDFILE2
Closing file UDFILE1
Creating file "UDFILE2" as Type 30.
Creating file "D_UDFILE2" as Type 3, Modulo 1, Separation 2.
Added "@ID", the default record for RetrieVe, to "D_UDFILE2".
Opening file UDFILE2
ora>SELECT  DISTINCT COLM, COLM+3, COLM*7 FROM TEDTAB7;
COLM......   COLM+3....   COLM*7....

         0                        3                    0
         1                        4                    7
         4                           7                       28
         6                           9                       42
4 records listed.
```

Next, exit `CONNECT` and enter two SELECT statements, one on UDFILE1 and one on UDFILE2:

```
ora> .Q
Disconnecting from 'ora' database
>SELECT * FROM UDFILE1;
COLC......   COLD......
```

```
detroit             lions
pittsburgh      steelers
new york        giants

3 records listed.
>SELECT * FROM UDFILE2;
COLM......   COLM+3....   COLM*7....

                    6            9           42
                0           3            0
                    1            4            7
                    4            7           28

4 records listed.
>
```

Using block mode

In the following example, the `.B` command enables block mode. The user enters a multiline SELECT statement, terminating it with the string GO instead of a semicolon.

```
syb> .b go
syb>select * from
empsSQL+
SQL+where deptno <300
SQL+go
    empno               lname        fname          deptno
---------  ----------   ----------  ---------
        17543           Smith        George            301
        23119           Brown        George            307
2 rows selected
```

The next example enables block mode and creates a stored procedure on an Oracle database:

```
ora> .B ON
ora> CREATE PROCEDURE sal_raise (emp_id IN NUMBER,
SQL+sal_incr IN NUMBER) AS
SQL+BEGIN
SQL+UPDATE emp
SQL+SET sal = sal + sal_incr
SQL+WHERE empno = emp_id;
SQL+IF SQL%NOTFOUND THEN
SQL+raise_application_error (-20011, 'Invalid Employee Number:'||
SQL+TO_CHAR (emp_id));
SQL+END IF;
SQL+END sal_raise;
SQL+<Return>
ora>
```

Chapter 4: Using UniData BCI

UniBasic programs use UniData BCI to exchange data between a UniData client and a server data source. A data source is a network node and a DBMS on that node. For example, you might have an order entry system on the machine running UniData and want to post this information to a central database. You use UniData BCI to connect your applications to the data source and exchange data.

This chapter describes how to do the following:

- Establish a connection to a data source
- Execute SQL statements at the data source
- Execute procedures stored on the data source
- Terminate the connection

Establishing a connection to a data source

Before connecting to a data source, the application does two things:

- Creates an UniData BCI environment, in which all connections to ODBC data sources are established.
- Creates one or more connection environments. Each connection environment supports a connection to a single data source.

Allocating the environment

The UniData BCI environment is a data structure that provides the context for all future connections to ODBC data sources. You allocate the environment with the `SQLAllocEnv` function. This function allocates memory for an environment and returns its address in a variable. This variable is the environment *handle*. You can allocate more than one environment at a time.

Allocating the connection environment

The connection environment is a data structure that supports a connection to a single data source. One UniData BCI environment can support many connection environments. You allocate the connection environment with the `SQLAllocConnect` function. This function allocates memory for a connection environment and returns its address, or handle, in a variable. Use this variable when you refer to a specific connection. You can establish more than one connection environment at a time.

Connecting to a data source

Use `SQLConnect` to establish a session between your application and the DBMS on the server.

When connecting to an ODBC server, `SQLConnect` contains the name of the data source and information needed to log in to the data source. After the connection is established, you can allocate an SQL statement environment to prepare to process SQL statements.

Processing SQL statements

After you allocate an SQL statement environment, you execute SQL statements.

Allocating the SQL statement environment

The SQL statement environment is a data structure that provides a context for delivering SQL statements to the data source, receiving data from the data source row by row, and sending data to the data source. An SQL statement environment belongs to one connection environment. You allocate the SQL statement environment with the `SQLAllocStmt` function. This function allocates memory for an SQL statement environment and returns its address, or handle, in a variable.

You can establish more than one SQL statement environment for the same connection environment.

Executing SQL statements

You can execute SQL statements in two ways:

- Direct execution
- Prepared execution

Note: Valid SQL statements can be either SQL statements supported by the DBMS or SQL statements conforming to the ODBC grammar specification as documented in the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Executing SQL statements directly

Direct execution is the simplest way to execute an SQL statement. Use the `SQLExecDirect` function to execute an SQL statement once, or when your program does not need information about the column results before executing the statement.

Preparing and executing SQL statements

Use prepared execution when you want to execute the same SQL statement more than once, or when you need information about SQL statement results before the statement is executed. Use the `SQLPrepare` and `SQLExecute` functions for prepared execution. `SQLPrepare` prepares the SQL statement once, then `SQLExecute` is called each time the SQL statement is to be executed.

For example, if you are inserting many data rows in a table, use the `SQLPrepare` function once, then use one `SQLExecute` for each row you insert. Before each `SQLExecute` call, set new values for the data to insert. To set new data values, you use parameter markers in your SQL statement (see Using parameter markers in SQL statements, on page 23). Using the `SQLPrepare` and `SQLExecute` functions in this way is more efficient than using separate `SQLExecDirect` calls, one for each row.

Using parameter markers in SQL statements

You can use parameter markers in SQL statements to mark the place where you will insert values to send to the data source. If your SQL statements contain parameter markers, you must call

SQLBindParameter once for each marker, to specify where to find the current value for each marker. For example, assume you create a table on the data source with the following command:

```
CREATE TABLE STAFF
   ( EMPNUM CHAR(3) NOT NULL,
   EMPNAME CHAR(20),
   GRADE INT,
CITY CHAR(15))
```

To insert data into this table, you might use SQLExecDirect to execute a series of INSERT statements. For example:

```
STATUS = SQLExecDirect(STMTENV,"INSERT INTO STAFF VALUES ('E9', 'Edward',
   10, 'Arlington')")
STATUS = SQLExecDirect(STMTENV,"INSERT INTO STAFF VALUES ('E10', 'John',
   12, 'Belmont')")
STATUS = SQLExecDirect(STMTENV,"INSERT INTO STAFF VALUES ('E11', 'Susan',
   13, 'Lexington')")
STATUS = SQLExecDirect(STMTENV,"INSERT INTO STAFF VALUES ('E12', 'Janet',
13, 'Waltham')")
```

The SQLExecDirect function takes two input variables. The first, STMTENV, is the name of the SQL statement environment. The second is the INSERT statement to be sent to the data source for execution.

Using several SQLExecDirect calls to insert data in this way is relatively slow. A better way to do this is to prepare the following INSERT statement for execution:

```
SQL = "INSERT INTO STAFF VALUES ( ?, ?, ?, ? )"
STATUS = SQLPrepare(STMTENV,SQL)
```

Each question mark in the statement is a parameter marker representing a value to be obtained from the application program when you execute the statement. You use the SQLBindParameter function to tell UniData BCI where to find the variables that will resolve each question mark in the statement. When you issue the SQLExecute call, UniData BCI picks up the variables you provided for these parameter markers, performs any required data conversions, and sends them to the data source, which executes the SQL statement with the new values.

Before you execute this statement, use SQLBindParameter calls to tell UniData BCI where to find the parameter values to use in the statement. For example:

```
STATUS = SQLBindParameter(STMTENV, 1, SQL.B.BASIC, SQL.CHAR, 3, 0, EMPNUM)
STATUS = SQLBindParameter(STMTENV, 2, SQL.B.BASIC, SQL.CHAR, 20, 0, EMPNAME)
STATUS = SQLBindParameter(STMTENV, 3, SQL.B.BASIC, SQL.INTEGER, 0, 0, GRADE)
STATUS = SQLBindParameter(STMTENV, 4, SQL.B.BASIC, SQL.CHAR, 15, 0, CITY)
STATUS = SQLPrepare(STMTENV, "INSERT INTO STAFF VALUES ( ?, ?, ?, ? )")
   .
   .
   .
EMPNUM = 'E9'
EMPNAME = 'Edward'
GRADE = 10
CITY = 'Arlington'
STATUS = SQLExecute(STMTENV)
EMPNUM = E10'
EMPNAME = 'John'
GRADE = 12
CITY = 'Belmont'
STATUS = SQLExecute(STMTENV)
   .
   .
```

.

The SQLBindParameter function takes seven input variables. The first, STMTENV, is the name of the SQL statement environment. The second is the number of the parameter marker in the INSERT statement to be sent. The third (SQL.B.BASIC) and fourth (SQL.CHAR, SQL.INTEGER) specify data types, used to convert the data from UniBasic to SQL data types. The fifth and sixth specify the parameter precision and scale. The seventh is the name of the variable that will contain the value to use in the INSERT statement.

You can also use parameter markers with SELECT statements when you want to specify variable conditions for queries. For example, you might use the following statements to select rows from STAFF when CITY is a variable loaded from your application:

```
STATUS = SQLBindParameter(STMTENV, 1, SQL.B.BASIC, SQL.CHAR, 15, 0, CITY)
STATUS = SQLPrepare(STMTENV, "SELECT * FROM STAFF WHERE CITY = ?")
PRINT "ENTER CITY FOR QUERY":
INPUT CITY
STATUS = SQLExecute(STMTENV)
```

Processing output from SQL statements

Once you execute an SQL statement at the data source, you can issue calls that tell you more about the results, and allow you to bring results from the data source back to your application.

You use the SQLNumResultCols function to find out how many columns the SQL statement produced in the result set. If it finds columns, you can use SQLDescribeCol or SQLColAttributes to get information about a column, such as its name, and the SQL data type it contains.

You use the SQLRowCount function to find out if the SQL statement changed any rows in the table. For example, if an SQL UPDATE statement changes 48 rows, SQLRowCount returns 48.

If an SQL statement produces a set of results at the data source, we say that a cursor is opened on the result set. You can think of this cursor as a pointer into the set of results, just as a cursor on a screen points to a particular line of text. An open cursor implies that there is a set of results pending at the data source.

To bring the results of the SQL statement from the data source to your application, use the SQLBindCol and SQLFetch functions. You use SQLBindCol to tell UniData BCI where to put data from a specific column and what application data type to store it as. For example, to print rows from the STAFF table, you might write the following:

```
SQLBindCol(STMTENV, 1, SQL.B.DEFAULT, EMPNUM)
SQLBindCol(STMTENV, 2, SQL.B.DEFAULT, EMPNAME)
SQLBindCol(STMTENV, 3, SQL.B.DEFAULT, EMPGRADE)
SQLBindCol(STMTENV, 4, SQL.B.DEFAULT, EMPCITY)

LOOP
WHILE STATUS <> SQL.NO.DATA.FOUND DO
   STATUS = SQLFetch(STMTENV)
   IF STATUS <> SQL.NO.DATA.FOUND
   THEN
      PRINT EMPNUM form:EMPNAME form:EMPGRADE form:EMPCITY
   END
REPEAT
```

The SQLBindCol function takes four input variables. The first, STMTENV, is the name of the SQL statement environment. The second is the number of the column. The third specifies the data type to

which to convert the incoming data. The fourth is the name of the variable where the column value is stored.

For each column, a call to `SQLBindCol` tells UniData BCI where to put each data value when you issue `SQLFetch`. Each `SQLFetch` stores the next row of data in the specified variables. Normally you fetch data rows until the end-of-data status flag is returned to the `SQLFetch` call.

The SQL.B.DEFAULT argument to `SQLBindCol` lets the SQL data type of the result column determine the UniBasic data type to which to convert data from the data source. For information about converting data, see Data conversion, on page 74.

For other examples showing how to execute SQL statements, see UniData BCI demo program, on page 82.

Freeing the SQL statement environment

Once all processing of an SQL statement is complete, use `SQLFreeStmt` to free resources in an SQL statement environment. Use one of the following options in the `SQLFreeStmt` call:

- SQL.CLOSE closes any open cursor associated with an SQL statement environment and discards any pending results. The SQL statement environment can then be reused by executing another SQL statement with the same or different parameters and bound column variables. SQL.CLOSE releases all locks held by the data source.
- SQL.UNBIND releases all bound column variables set by `SQLBindCol` for the SQL statement environment.
- SQL.RESET.PARAMS releases all parameter marker variables set by `SQLBindParameter` for the SQL statement environment.
- SQL.DROP releases the SQL statement environment, frees all resources, closes any cursor, and cancels all pending results. All column variables are unbound and all parameter marker variables are reset. This option terminates access to the SQL statement environment.

For example, the following statement frees the SQL statement environment in the demo program (see UniData BCI demo program, on page 82):

```
STATUS = SQLFreeStmt(STMTENV, SQL.DROP)
```

Terminating the connection

When your program is ready to terminate the connection to the data source, it should execute the following tasks:

- Disconnect from the data source
- Release the connection environment
- Release the SQL Client Interface environment

Use `SQLDisconnect` to close the connection associated with a specific connection environment. All active transactions must be committed or rolled back before disconnecting (see Transaction management, on page 28). If there are no transactions pending, `SQLDisconnect` frees all allocated SQL statement environments.

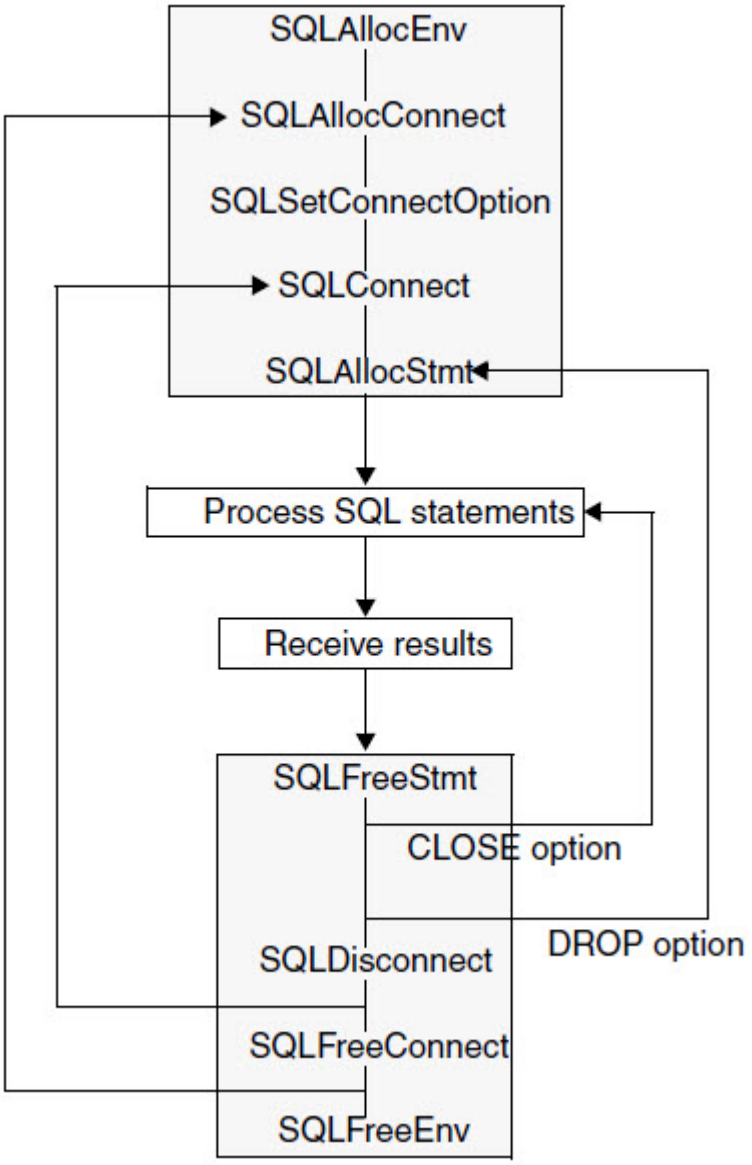
Use `SQLFreeConnect` to release a connection environment and its associated resources. The connection environment must be disconnected from the data source before you use this call, or an error occurs.

Use `SQLFreeEnv` to release the UniData BCI environment and all resources associated with it. Disconnect all sessions with the `SQLDisconnect` and `SQLFreeConnect` functions before you use `SQLFreeEnv`.

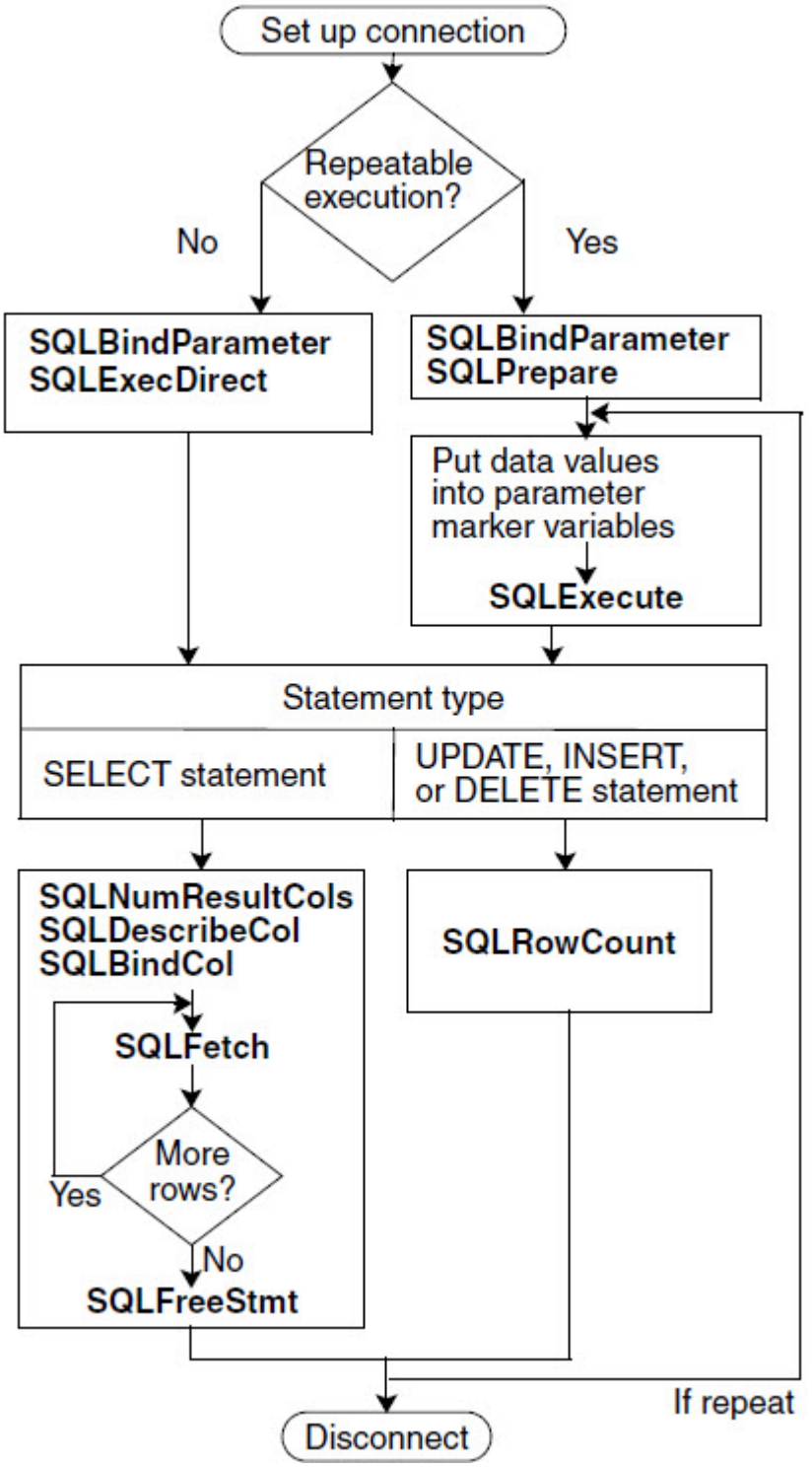
In the demo program (see UniData BCI demo program, on page 82), the following statements close the connection to the data source and free the connection and UniData BCI environments:

```
STATUS = SQLDisconnect(CONENV)
STATUS = SQLFreeConnect(CONENV)
STATUS = SQLFreeEnv(DBCENV)
```

The following illustration shows some function calls used in a BASIC application.



The next illustration shows the order of function calls you use to execute a simple SQL statement.



Transaction management

There are two ways to control transaction processing with UniData BCI, private transactions or nonprivate transactions.

Private transactions

A private transaction is one that is fully managed by the DBMS on the server.

The UniBasic application sends transaction management statements using the UniData BCI API to the data source. For more information about these transaction management functions, see UniData BCI functions, on page 31.

Nonprivate transactions

A nonprivate transaction is fully managed by UniBasic. Use the UniBasic statements `TRANSACTION START`, `TRANSACTION COMMIT`, or `TRANSACTION ABORT` to provide transaction management control in your application.

- Outside a transaction, changes are committed immediately (autocommit mode).
- Use `TRANSACTION START` to put all current connections into manual commit mode. Connections established within a transaction are also in manual commit mode.
- Use `TRANSACTION COMMIT` or `TRANSACTION ABORT` to terminate the transaction.

Note: In nonprivate transactions, your application programs should not try to issue transaction control statements directly to the data source (for instance, by issuing a `COMMIT` statement with `SQLExecDirect` or `SQLPrepare`). Programs should use only UniBasic transaction control statements. UniData issues the correct combination of transaction control statements and middleware transaction control function calls that are appropriate for the DBMS you are using. Trying to use `SQLExecDirect` and `SQLExecute` to execute explicit transaction control statements on ODBC data sources can cause unexpected results and errors.

Distributed transactions

A distributed transaction is a transaction that updates more than one data source. Be careful when you use distributed transactions. The UniData transaction manager does not support the two-phase commit protocol that ensures that all operations are either committed or rolled back properly. If a `COMMIT` fails, the systems involved may be out of sync, since the local part of the `COMMIT` can succeed even though the remote part fails.

If your program uses distributed transactions, you must ensure that it can recover from a `COMMIT` failure, or that enough information is available to let you manually restore the databases to a consistent state.

Detecting errors

Errors can be detected by UniData BCI, by the UniRPC, by the ODBC driver, or by the data source.

For more information, see the `SQLError` function in UniData BCI functions, on page 31 and Error codes, on page 90.

Chapter 5: Calling and executing procedures

Client programs can call and execute procedures that are stored on a database server. Procedures can accept and return parameter values and return results to the calling program.

Procedures let developers predefine database actions on the server. Procedures can provide a simple interface to users, insulating them from the names of tables and columns as well as from the syntax of SQL. Procedures can enforce additional database rules beyond simple referential integrity and constraints. Such rules need not be coded into each application that references the data, providing consistency and easy maintenance.

Procedures can provide a significant performance improvement in a client/server environment. Applications often have many steps, where the result from one step becomes the input for the next. If you run such an application from a client, it can take a lot of network traffic to perform each step and get results from the server. If you run the same program as a procedure, all the intermediate work occurs on the server; the client simply calls the procedure and receives a result.

Calling and executing ODBC procedures

A UniBasic client program that is connected to an ODBC data source can call and execute procedures stored on the server, provided that the ODBC driver and the database on the server support a procedure call mechanism. The standard ODBC grammar for a procedure call statement is:

```
{ call procedure [ ( [ parameter [ , parameter] … ] ) ] }
```

As when calling UniData procedures, you must execute (not simply prepare) a call statement before you can successfully use any of the following functions to inquire about the results of the procedure:

- `SQLNumResultCols`
- `SQLColAttributes`
- `SQLRowCount`

Chapter 6: UniData BCI functions

This chapter describes the UniData BCI functions in alphabetical order.

Overview

The following table lists the UniData BCI functions according to how they are used.

Category	Functions
Initializing	SQLAllocEnv SQLAllocConnect SQLSetConnectOption SQLConnect SQLAllocStmt SQLPrepare
Exchanging data	SQLTransact SQLBindParameter SQLSetParam SQLExecute SQLExecDirect SQLRowCount SQLNumResultCols SQLDescribeCol SQLColAttributes SQLBindCol SQLFetch SQLGetData SQLGetInfo SQLGetTypeInfo SQLNumParams SQLParamOptions SQLTables
Exchanging data	SQLColumns SQLSpecialColumns SQLStatistics
Processing errors	SQLError

Category	Functions
Disconnecting	SQLFreeStmt SQLCancel SQLDisconnect SQLFreeConnect SQLFreeEnv

The syntax diagram for each function includes the function name and any applicable input, output, and return variables. For example:

```
status = SQLAllocConnect (bci.env, connect.env)
```

You must call the SQLAllocEnv function before you use any other UniData BCI function.

Variable names

In the previous syntax diagram, *status* is a return variable that the function returns upon completion. *bci.env* is an input variable whose value is provided by a previous function call. *connect.env* is an output variable containing a value output by the function. Names of return variables, input variables, and output variables are user-defined.

Return values

UniData BCI functions return a value to the *status* variable.

Return values are the following:

Return value	Description
0 – SQL.SUCCESS	Function completed successfully.
1 – SQL.SUCCESS.WITH.INFO	Function completed successfully with a possible nonfatal error. Your program can call SQLError to get information about the error.
-1 – SQL.ERROR	Function failed. Your program can call SQLError to get information about the error.
-2 – SQL.INVALID.HANDLE	Function failed because the environment, connection, or SQL statement variable is invalid.
100 – SQL.NO.DATA.FOUND	All rows from the result set were fetched (SQLFetch), or no error to report (SQLError).

Error codes

Any SQL Client Interface function call can generate errors. Use the SQLError function after any other function call for which the returned status indicates an error condition.

For a list of UniData BCI error codes, see Error codes, on page 90.

SQLAllocConnect

`SQLAllocConnect` allocates and initializes a connection environment in a UniData BCI environment.

Use this function to create a connection environment to connect to a particular data source. One UniData BCI environment can have several connection environments, one for each data source. The function stores the internal representation of the connection environment in the *connect.env* variable.

> **Note:** Use the connection environment variable only in UniData BCI calls that require it. Using it improperly can cause a runtime error and break communication with the data source.

Syntax

```
status = SQLAllocConnect (bci.env, connect.env)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
bci.env	UniData BCI environment variable returned in an `SQLAllocEnv` call.
connect.env	Variable that represents the allocated connection environment.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLAllocEnv

`SQLAllocEnv` creates an environment in which to execute UniData BCI calls.

Use this function to allocate memory for a UniData BCI environment. The function stores the address in the *bci.env* variable. `SQLAllocEnv` must be the first UniData BCI call issued in any application.

You can allocate more than one UniData BCI environment.

> **Note:** Use the environment variable only in UniData BCI calls that require it. Using it in any other context causes a runtime error or breaks communication with the data source.

Syntax

```
status = SQLAllocEnv (bci.env)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
bci.env	Variable that represents the allocated UniData BCI environment.

Return values

The following table describes return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR

SQLAllocStmt

`SQLAllocStmt` creates an SQL statement environment in which to execute SQL statements.

Use this function to allocate memory for an SQL statement environment.

Note: Use the SQL statement environment variable only in UniData BCI calls that require it. Using it in any other context causes a runtime error or breaks communication with the data source.

Syntax

```
status = SQLAllocStmt (connect.env, statement.env)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
connect.env	Connection environment used in `SQLAllocConnect` and `SQLConnect` calls. If you have not established a connection to the data source using *connect.env*, an error is returned to the application.
statement.env	Variable that represents an SQL statement environment.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLBindCol

Use this function to tell UniData BCI where to return the results of an `SQLFetch` call. `SQLBindCol` defines the name of the variable (*column*) to contain column results retrieved by `SQLFetch`, and specifies the data conversion (*data.type)* on the fetched data. `SQLBindCol` has no effect until `SQLFetch` is used.

Normally you call SQLBindCol once for each column of data in the result set. When SQLFetch is issued, data is moved from the result set at the data source and put into the variables specified in the SQLBindCol call, overwriting existing variable contents.

Data is converted from the SQL data type at the data source to the UniBasic data type requested by the SQLBindCol call, if possible. If data cannot be converted to *data.type*, an error occurs. For information about data conversion types, see Data conversion, on page 74.

Values are returned only for bound columns. Unbound columns are ignored and are not accessible. For example, if a SELECT command returns three columns, but SQLBindCol was called for only two columns, data from the third column is not accessible to your program. If you bind more variables than there are columns in the result set, an error is returned. If you bind no columns and an SQLFetch is issued, the cursor advances to the next row of results.

You need not use SQLBindCol with SQL statements that do not produce result sets.

Syntax

```
status = SQLBindCol (statement.env, col#, data.type, column)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment of the executed SQL statement.
col#	Column number of result data, starting at 1. This value must be from 1 to the number of columns returned in an operation.
data.type	BASIC data type into which to convert the incoming data. Possible values are the following: ▪ SQL.B.CHAR – Character string data. ▪ SQL.B.BINARY – Bit string (raw) data. ▪ SQL.B.NUMBER – Numeric data (integer or double). ▪ SQL.B.DEFAULT – SQL data type determines the BASIC data type. For information about data conversion, see Data conversion, on page 74 ▪ SQL.B.INTDATE – UniData date in internal format. ▪ SQL.B.INTTIME – UniData time in internal format.
column	Variable that will contain column results obtained with SQLFetch.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLBindParameter

SQLBindParameter specifies where to find values for input parameter markers when you issue an SQLExecute or SQLExecDirect call. For output parameter markers, SQLBindParameter specifies where to find the return value of a called procedure.

Syntax

```
status = SQLBindParameter ( statement.env, mrk#, data.type, sql.type,
prec, scale, param [ , param.type ] )
```

Description

Parameter markers are placeholders in SQL statements. Input parameter markers let a program send user-defined values with the SQL statement when an SQLExecute or SQLExecDirect call is executed repeatedly. Output parameter markers receive values returned from a called procedure. The placeholder character is ? (question mark). For more information about parameter markers, see Using parameter markers in SQL statements, on page 23.

SQLBindParameter tells the system where to find the variables to substitute for parameter markers in the SQL statement and how to convert the data before sending it to the data source. You need to do only one SQLBindParameter for each parameter marker in the SQL statement, no matter how many times the statement is to be executed.

For example, consider the following SQL statement:

```
INSERT INTO T1 VALUES (?,?,?);
```

If you want to load 1000 rows, you need issue only three SQLBindParameter calls, one for each question mark.

Normally you specify *data.type* as SQL.B.BASIC. If you specify *sql.type* as SQL.DATE, however, you can specify *data.type* as SQL.B.INTDATE; if you specify *sql.type* as SQL.TIME, you can specify *data.type* as SQL.B.INTTIME. If you specify *sql.type* as SQL.BINARY, SQL.VARBINARY, or SQL.LONGVARBINARY, you can specify *data.type* as SQL.B.BINARY.

If you use SQL.B.INTDATE, the UniData BCI assumes the program variable holds a date in UniData internal date format and uses the DATEFORM conversion string to convert the internal date to an external format as required by the data source. To set or change the DATEFORM conversion string, see the SQLSetConnectOption function. For details about date and time conversions, see Data conversion, on page 74.

If you specify *sql.type* as SQL.TIME and *data.type* as SQL.B.INTTIME, UniData BCI assumes the program variable holds a time in UniData internal time format and does not convert the data.

SQLBindParameter uses the value of *prec* only for the following SQL data types:

- SQL.CHAR
- SQL.VARCHAR
- SQL.LONGVARCHAR
- SQL.WCHAR
- SQL.WVARCHAR
- SQL.WLONGVARCHAR
- SQL.BINARY
- SQL.VARBINARY
- SQL.LONGVARBINARY

- SQL.NUMERIC
- SQL.DECIMAL

For all other data types, the extended parameters DBLPREC, FLOATPREC, and INTPREC determine the maximum length for strings representing double-precision numbers, floating-point numbers, and integers.

Parameters

The following table describes each parameter of the syntax.

Input variable	Description
statement.env	SQL statement environment associated with an SQL statement.
mrk#	Number of the parameter marker in the SQL statement to which this call refers. Parameter markers in the SQL statement are numbered left to right, starting at 1.
data.type	UniBasic data type to bind to the parameter. *data.type* must be one of the following: ▪ SQL.B.BASI – Use with any *sql.type*. ▪ SQL.B.BINARY – Use only when *sql.type* is SQL.BINARY, SQL.VARBINARY, or SQL.LONGVARBINARY. ▪ SQL.B.INTDATE – Use only when *sql.type* is SQL.DATE. ▪ SQL.B.INTTIME – Use only when *sql.type* is SQL.TIME.
sql.type	SQL data type to which the UniBasic variable is converted. For information about converting UniBasic data to SQL data types, see Converting UniBasic data to SQL data, on page 76.
prec	Precision of the parameter, representing the width of the parameter. If *prec* is 0, default values are used.
scale	Scale of the parameter, used only when *sql.type* is SQL.DECIMAL or SQL.NUMERIC.
param	Variable that contains the data to use when `SQLExecute` or `SQLExecDirect` is called.
param.type	Type of parameter. *param.type* can be one of the following: ▪ SQL.PARAM.INPUT – Use for parameters in an SQL statement that does not call a procedure, or for input parameters in a procedure call. ▪ SQL.PARAM.OUTPUT – Use for parameters that mark the output parameter in a procedure. ▪ QL.PARAM.INPUT.OUTPUT – Use for an input/output parameter in a procedure. If you do not specify *param.type*, SQL.PARAM.INPUT is used.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR

Return value	Description
-2	SQL.INVALID.HANDLE

SQLCancel

This function is equivalent to the `SQLFreeStmt` call with the SQL.CLOSE option. It closes any open cursor associated with the SQL statement environment and discards pending results at the data source.

It is good practice to issue `SQLCancel` when all results have been read from the data source, even if the SQL statement environment will not be reused immediately for another SQL statement. Issuing `SQLCancel` frees any locks that may be held at the data source.

Syntax

status = **SQLCancel** (*statement.env*)

Parameters

The following table describes each parameter of the syntax.

Input variable	Description
statement.env	SQL statement environment.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLColAttributes

Use this function to get information about a column. `SQLColAttributes` returns the specific information requested by the value of *col.attribute*.

Some DBMSs (such as SYBASE) do not make column information available until after the SQL statement is executed. In such cases, issuing an `SQLColAttributes` call before executing the statement produces an error.

The SQL.SUCCESS.WITH.INFO return occurs when you issue the call for a column that contains an unsupported data type or when *text.var* is truncated. The SQL data type returned is SQL.BINARY (-2).

If you are connected to an ODBC database, SQL.COLUMN.NULLABLE always returns SQL.NULLABLE.UNKNOWN.

When you are connected to an ODBC data source, calling `SQLColAttributes` with one of the column attributes returns a status of SQL.ERROR with SQLSTATE set to S1091.

Syntax

```
status = SQLColAttributes (statement.env, col#, col.attribute,
text.var, num.var)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment of the executed SQL statement.
col#	Column number to describe, starting with 1.
col.attribute	Attribute of the column that needs information. *col.attribute* values are listed above the Syntax section. These values are defined in the ODBC.H file. The ODBC.H file, on page 92 lists the contents of the ODBC.H file.
text.var	Contains column information for attributes returning text data. If the return value is numeric, *text.var* contains invalid information.
num.var	Contains column information for attributes returning numeric data. If the return value is text, *num.var* contains invalid information.

The following table lists the column attributes you can use with ODBC databases.

Column attribute	Output	Description
SQL.COLUMN.AUTO.INCREMENT	*num.var*	▪ 1 – TRUE if the column values are incremented automatically. ▪ 0 – FALSE if the column values are not incremented automatically.
SQL.COLUMN.CASE.SENSITIVE	*num.var*	▪ 1 – TRUE for character data. ▪ 0 – FALSE for all other data.
SQL.COLUMN.COUNT	*num.var*	Number of columns in result set. The *col#* argument must be a valid column number in the result set.
SQL.COLUMN.DISPLAY.SIZE	*num.var*	Maximum number of characters required to display data from the column.
SQL.COLUMN.LABEL	*text.var*	Column heading.
SQL.COLUMN.LENGTH	*num.var*	Number of bytes transferred by an `SQLFetch` call.
SQL.COLUMN.NAME	*text.var*	Name of specified column.

Column attribute	Output	Description
SQL.COLUMN.NULLABLE	*num.var*	Column can contain null values. Can return one of the following: ▪ 0 – SQL.NO.NULLS ▪ 1 – SQL.NULLABLE ▪ 2 – SQL.NULLABLE. UNKNOWN
SQL.COLUMN.PRECISION	*num.var*	Column precision.
SQL.COLUMN.SCALE	*num.var*	Column scale.
SQL.COLUMN.SEARCHABLE	*num.var*	Always returns 3, SQL.SEARCHABLE.
SQL.COLUMN.TABLE.NAME	*text.var*	Name of the table to which the column belongs. If the column is an expression, an empty string is returned.
SQL.COLUMN.TYPE	*num.var*	Number representing the SQL type of this column. See The ODBC.H file, on page 92 for data type definitions. See Data conversion, on page 74 for a list of data types.
SQL.COLUMN.TYPE.NAME	*text.var*	Data type name for column, specific to the data source.
SQL.COLUMN.UNSIGNED	*num.var*	▪ 1 – TRUE for nonnumeric data types. ▪ 0 – FALSE for all other data types.
SQL.COLUMN.UPDATABLE	*num.var*	Any expressions or computed columns return SQL.ATTR.READONLY, and stored data columns return SQL.ATTR.WRITE.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

ODBC data sources

The following table lists the column attributes you can use only with ODBC databases.

Column attribute	Output	Description
SQL.COLUMN.MONEY	*num.var*	▪ 1 – TRUE if column is money data type. ▪ 0 – FALSE if column is not money data type.
SQL.COLUMN.OWNER.NAME	*text.var*	Owner of the table containing the column.
SQL.COLUMN.QUALIFIER.NAME	*text.var*	Qualifier of the table containing the column.

SQLColumns

This function returns a result set in *statement.env* as a cursor of 12 columns describing those columns found by the search pattern (see `SQLTables`). As with `SQLTables`, the search is done on the SQL catalog. This is a standard result set that can be accessed with `SQLFetch`. The ability to obtain descriptions of columns does not imply that a user has any privileges on those columns.

Syntax

```
status = SQLColumns (statement.env, schema, owner, tablename,
columnname )
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment.
schema	Schema name search pattern.
owner	Table owner number search pattern.
tablename	Table name search pattern.
columnname	Column name search pattern.

Result set

The result set contains 12 columns:

Column name	Data type
TABLE.SCHEMA	VARCHAR(128)
OWNER	INTEGER
TABLE.NAME	VARCHAR(128)
COLUMN.NAME	VARCHAR(128)
DATA.TYPE	SMALLINT
TYPE.NAME	VARCHAR(128)
NUMERIC.PRECISION	INTEGER
CHAR.MAX.LENGTH	INTEGER
NUMERIC.SCALE	SMALLINT
NUMERIC.PREC.RADIX	SMALLINT
NULLABLE	SMALLINT

Column name	Data type
REMARKS	VARCHAR(254)

The application is responsible for binding variables for the output columns and fetching the results using `SQLFetch`.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLSTATE values

The following table shows all possible SQLSTATE values returned by `SQLColumns`.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1008	Cancelled. Execution of the statement was stopped by an `SQLCancel` call.
S1010	Function sequence error. The *statement.env* specified is currently executing an SQL statement.
S1C00	The table owner field was not numeric.
24000	Invalid cursor state. Results are still pending from the previous SQL statement. Use `SQLCancel` to clear the results.
42000	Syntax error or access violation. This can be caused by a variety of reasons. The native error code returned by the `SQLError` call indicates the specific UniData error that occurred.

SQLConnect

Use this function to connect to the ODBC data source specified by *data.source*. Use the *login1* and *login2* parameters to log in to the DBMS specified by the ODBC *data.source*.

You cannot use `SQLConnect` within a transaction. An `SQLConnect` call issued within a transaction returns SQL.ERROR, and sets SQLSTATE to 25000, indicating that the `SQLConnect` function is illegal within a transaction.

A connection is established when the data source validates the user name and authorization.

Syntax

```
status = SQLConnect (connect.env, data.source, login1, login2)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
connect.env	Connection environment assigned in a previous `SQLAllocConnect`.
data.source	Data source name. For ODBC data sources, this is the name of a data source specified by the data source management program you are using.
login1	For ODBC data sources, this is typically the password for the remote database or operating system. For the specific information required for *login1* when connecting to ODBC data sources, see the configuration for the specific driver used.
login2	For ODBC data sources, this is typically the password for the remote database or operating system. For the specific information required for *login2* when connecting to ODBC data sources, see the configuration for the specific driver used.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLDescribeCol

Use this function to get information about the column described by *col#*.

The SQL.SUCCESS.WITH.INFO return occurs when you issue the call for a column that contains an unsupported data type, or if *col.name* is truncated. The SQL data type returned is SQL.BINARY (-2).

Syntax

```
status = SQLDescribeCol (statement.env, col#, col.name, sql.type, prec,
scale, null)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment of the executed SQL statement.
col#	Column number to describe, starting with 1.
col.name	Column name.
sql.type	SQL data type of the column, a numeric code defined in the ODBC.H file. See The ODBC.H file, on page 92 for more information.
prec	Precision of the column, or –1 if precision is unknown.
scale	Scale of the column, or –1 if scale is unknown.

Parameter	Description
null	One of the following: ▪ 0 – SQL.NO.NULLS: field cannot contain NULL. ▪ 1 – SQL.NULLABLE: field can contain NULL. ▪ 2 – SQL.NULLABLE.UNKNOWN: not known whether field can contain NULL.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLDisconnect

`SQLDisconnect` disconnects a connection environment from a data source.

You cannot use `SQLDisconnect` within a transaction. An `SQLDisconnect` call issued within a transaction returns SQL.ERROR, and sets SQLSTATE to 25000. You must commit or abort active transactions before disconnecting, and you must be in autocommit mode. If there is no active transaction, `SQLDisconnect` frees all SQL statement environments owned by this connection before disconnecting.

`SQLDisconnect` returns SQL.SUCCESS.WITH.INFO if an error occurs but the disconnect succeeds.

Syntax

```
status = SQLDisconnect (connect.env)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
connect.env	Connection environment.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLError

`SQLError` returns error status information about one of the three environments you use.

Use `SQLError` when a function returns a status value other than SQL.SUCCESS or SQL.INVALID.HANDLE. `SQLError` returns a value in *sqlstate* when UniData BCI detects an error condition. The *dbms.code* field contains information from the data source that identifies the error.

Each environment type maintains its own error status. `SQLError` returns errors for the right-most nonnull environment. For example, to get errors associated with a connection environment, specify input variables and constants in the following order:

bci.env, *connect.env*, SQL.NULL.HSTMT

To get errors associated with a particular SQL statement environment, specify the following:

bci.env, *connect.env*, *statement.env*

If all arguments are null, `SQLError` returns a status of SQL.NO.DATA.FOUND and sets SQLSTATE to 00000.

Since multiple errors can be returned for a variable, you should call `SQLError` until it returns a status of SQL.NO.DATA.FOUND. This ensures that all errors are reported.

Syntax

```
status = SQLError (bci.env, connect.env, statement.env, sqlstate,
dbms.code, error)
```

ODBC data sources

When a program is connected to an ODBC server, errors can be detected by UniData BCI, by the ODBC driver, or by the data source. When the error is returned, the source of the error is indicated by bracketed items in the *error* output variable, as shown in the following examples.

Errors detected by UniData BCI:

```
[U2][SQL Client] An illegal configuration option was found
```

For information about errors detected by the UniData BCI, see Error codes, on page 90.

Errors detected by the ODBC driver:

```
SQLConnect error:   Status = -1   SQLState = S1000   Natcode = 9352
[ODBC] [INTERSOLV][ODBC Oracle driver][Oracle]ORA-09352: Windows 32-bit
Two-Task driver unable to spawn new ORACLE task
```

For information about errors detected by the ODBC driver manager, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Errors detected by the data source:

```
[U2][SQL Client][UniData] Database not found or no system permissions.
```

For information about errors detected by the data source, see the documentation provided for the DBMS running on the data source.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
bci.env	UniData BCI environment or the constant SQL.NULL.HENV.
connect.env	Connection environment or the constant SQL.NULL.HDBC.
statement.env	SQL statement environment or the constant SQL.NULL.HSTMT.
sqlstate	SQLSTATE code. This code describes the UniData BCI Client error associated with the environment passed. *sqlstate* is always a five-character string. For a list of SQLSTATE codes and their meanings, see Error codes, on page 90.
dbms.code	Error code specific to the data source. *dbms.code* contains an integer error code from the data source. If *dbms.code* is 0, the error was detected by UniData BCI. For the meanings of specific error codes, see the documentation provided for the data source.
error	Text describing the error in more detail.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE
100	SQL.NO.DATA.FOUND

SQLExecDirect

`SQLExecDirect` accepts an SQL statement or procedure call and delivers it to the data source for execution. It uses the current values of any SQL statement parameter markers.

`SQLExecDirect` differs from `SQLExecute` in that it does not require a call to `SQLPrepare`. `SQLExecDirect` prepares the SQL statement or procedure call implicitly. Use `SQLExecDirect` when you do not need to execute the same SQL statement or procedure repeatedly.

You can use parameter markers in the SQL statement or procedure call as long as you have resolved each marker with an `SQLBindParameter` call. For information about parameter markers, see Using parameter markers in SQL statements, on page 23.

After an `SQLExecDirect` call you can use `SQLNumResultCols`, `SQLDescribeCol`, `SQLRowCount`, or `SQLColAttributes` to get information about the resulting columns. You can use `SQLNumResultCols` to determine if the SQL statement or procedure call created a result set.

If the executed SQL statement or procedure produces a set of results, you must use an `SQLFreeStmt` call with the SQL.CLOSE option before you execute another SQL statement or procedure call using the same SQL statement environment. The SQL.CLOSE option cancels any pending results still waiting at the data source.

Your application programs should not try to issue transaction control statements directly to the data source (for instance, by issuing a COMMIT statement with `SQLExecDirect` or `SQLPrepare`). Programs should use only UniBasic transaction control statements. UniData BCI issues the correct combination of transaction control statements and middleware transaction control function calls that are appropriate for the DBMS you are using. Trying to use `SQLExecDirect` to execute explicit transaction control statements on ODBC data sources can cause unexpected results and errors.

When SQLExecDirect calls a procedure, it does not begin a transaction. If a transaction is active when a procedure is called, the current transaction nesting level is maintained.

Note: If you execute a stored procedure or enter a command batch with multiple SELECT statements, the results of only the first SELECT statement are returned.

Syntax

```
status = SQLExecDirect (statement.env, statement)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment from a previous SQLAllocStmt.
statement	Either an SQL statement or a call to an SQL procedure, to be executed at the data source. If you are connected to an ODBC data source, it may treat identifiers and keywords in the SQL statement case-sensitively. To call an SQL procedure, use one of the following syntaxes: [{] CALL *procedure* [([*parameter* [, *parameter*] …])] [}] If you are connected to an ODBC data source, use the first syntax and enclose the entire call statement in braces.
procedure	Name of the procedure. If the procedure name contains characters other than alphabetic or numeric, enclose the name in double quotation marks. To embed a single double quotation mark in the procedure name, use two consecutive double quotation marks.
parameter	Either a literal value or a parameter marker that marks where to insert values to send to or receive from the data source. Programmatic SQL uses a ? (question mark) as a parameter marker. Use parameters only if the procedure is a subroutine. The number and order of parameters must correspond to the number and order of the subroutine arguments. For an ODBC data source, parameters should be of the same data type as the procedure requires.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLExecute

Use this function to repeatedly execute an SQL statement, using different values for parameter markers. You must use an SQLPrepare call to prepare the SQL statement before you can use

SQLExecute. If the SQL statement specified in the SQLPrepare call contains parameter markers, you must also issue an SQLBindParameter call for each marker in the SQL statement before you use SQLExecute. After you load the parameter marker variables with data to send to the data source, you can issue the SQLExecute call. By setting new values in the parameter marker variables and calling SQLExecute, new data values are sent to the data source and the SQL statement is executed using those values.

If the SQL statement uses parameter markers, SQLExecute performs any data conversions required by the SQLBindParameter call for the parameter markers. See Data conversion, on page 74 for details.

If the SQL statement executed produces a set of results, you must use an SQLFreeStmt call with the SQL.CLOSE option before you execute another SQL statement using the same SQL statement environment. The SQL.CLOSE option cancels any pending results still waiting at the data source.

Your application programs should not try to issue transaction control statements directly to the data source (for instance, by issuing a TRANSACTION COMMIT statement with SQLExecDirect or SQLPrepare). Programs should use only UniBasic transaction control statements. UniData BCI issues the correct combination of transaction control statements and middleware transaction control function calls that are appropriate for the DBMS you are using. Trying to use SQLExecute to execute explicit transaction control statements on ODBC data sources can cause unexpected results and errors.

SQLExecute tells the data source to execute a prepared SQL statement or a called procedure, using the current values of any parameter markers used in the statement. Using SQLExecute with an SQLBindParameter call is the most efficient way to execute a statement repeatedly, since the statement does not have to be parsed by the data source each time it is issued.

Syntax

```
status = SQLExecute (statement.env)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment associated with a prepared SQL statement.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLFetch

Use this function to retrieve the next row's column values from the result set at the data source and put them into the variables specified with SQLBindCol. SQLFetch performs any required data conversions.

See Data conversion, on page 74 for details.

SQLFetch returns SQL.SUCCESS.WITH.INFO if numeric data is truncated or rounded when converting SQL values to UniData values.

SQLFetch logically advances the cursor to the next row in the result set. Unbound columns are ignored and are not available to the application. When no more rows are available, SQLFetch returns a status of 100 (SQL.NO.DATA.FOUND).

Your application must issue an SQLFetch call at the same transaction nesting level (or deeper) as the corresponding SQLExecDirect or SQLExecute call. Also, an SQLFetch call must be executed at the same transaction isolation level as the SELECT statement that generates the data. If it does not, SQLFetch returns SQL.ERROR and sets SQLSTATE to S1000.

Use SQLFetch only when a result set is pending at the data source.

Syntax

```
status = SQLFetch (statement.env)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment of the executed SQL statement.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE
1	SQL.SUCCESS.WITH.INFO
100	SQL.NO.DATA.FOUND

SQLFreeConnect

SQLFreeConnect releases a connection environment and its resources.

You must use SQLDisconnect to disconnect the connection environment from the data source before you release the connection environment with SQLFreeConnect, otherwise an error is returned.

Syntax

```
status = SQLFreeConnect (connect.env)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
connect.env	Connection environment.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLFreeEnv

`SQLFreeEnv` releases an SQL Client Interface environment and its resources.

You must use `SQLFreeConnect` to release all connection environments attached to the UniData BCI environment before you release the UniData BCI environment with `SQLFreeEnv`, otherwise an error is returned.

Syntax

```
status = SQLFreeEnv (bci.env)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
bci.env	UniData BCI environment.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLFreeStmt

`SQLFreeStmt` frees some or all resources associated with an SQL statement environment.

If your program uses the same SQL statement environment to execute different SQL statements, use `SQLFreeStmt` with the SQL.DROP option, then use `SQLAllocStmt` to reallocate a new SQL statement environment. This unbinds all bound columns and resets all parameter marker variables.

It is good practice to issue `SQLFreeStmt` with the SQL.CLOSE option when all results have been read from the data source, even if the SQL statement environment will not be reused immediately for

another SQL statement. Issuing `SQLFreeStmt` with the SQL.CLOSE option frees any locks that may be held at the data source.

Syntax

```
status = SQLFreeStmt (statement.env, option)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment.
option	*option* is one of the following: ▪ SQL.CLOSE – Closes any open cursor associated with the SQL statement environment and discards pending results at the data source. Using the SQL.CLOSE option cancels the current query. All parameter markers and columns remain bound to the variables specified in the `SQLBindCol` and `SQLBindParameter` calls. ▪ SQL.UNBIND – Releases all bound column variables defined in `SQLBindCol` for this SQL statement environment. ▪ SQL.RESET.PARAMS – Releases all parameter marker variables set by `SQLBindParameter` for this SQL statement environment. ▪ SQL.DROP – Releases the SQL statement environment. This option terminates all access to the SQL statement environment. SQL.DROP also closes cursors, discards pending results, unbinds columns, and resets parameter marker variables. Options are defined in the ODBC.H file. See The ODBC.H file, on page 92 for more information.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLGetData

The `SQLGetData` function retrieves data that exceeds the buffer space allocated for the data. It also retrieves data from columns in the result set that were not bound. This resolves a “Data Truncated" error message that may have been experienced with BCI in the past.

Syntax

```
status = SQLGetData(stmt.env, loc, type, retvar, len)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
stmt.env	The statement handle. [INPUT]
loc	Column number in the result set numbered left to right starting at 1. [INPUT]
type	The data type of the result for the column. BCI only supports SQL.CHAR for this parameter. [INPUT]
retvar	Variable to return the data. [OUTPUT]
len	Length to retrieve the data. [INPUT]

If the data in the column is null, retvar will be assigned to @NULL if the null value flag is on, or an empty string if it is not.

All unbound columns process with SQLGETDATA must have higher column numbers than the bound columns in the result set.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.SUCCESS.WITH.INFO
-2	SQL.NO.DATA.FOUND
	SQL.ERROR
	SQL.INVALID.HANDLE

Example

The following program illustrates a UniBasic program using SQLGETDATA:

```
$INCLUDE UD.INCLUDE ODBC.H

verbose = 1

datasource = "mydatasource" * ODBC datasouce name
username = "my user" * login user
passwd = "my password" * password

STMT = "SELECT RENTAL_PRICE, NAME FROM TAPES.TAPES" * example of SQL statement

FUNC="SQLALLOCENV"
status = SQLALLOCENV(bci.env)
IF status NE SQL.SUCCESS THEN PRINT "SQLALLOCENV failed"
IF verbose THEN CRT "status of ":FUNC:" is ":status

FUNC="SQLALLOCCONNECT"
status = SQLALLOCCONNECT(bci.env, connect.env)
IF status NE SQL.SUCCESS THEN PRINT "SQLALLOCCONNECT failed"
IF verbose THEN CRT "status of ":FUNC:" is ":status

FUNC="SQLCONNECT"
status = SQLCONNECT(connect.env, datasource, username, passwd
IF status LT SQL.SUCCESS THEN PRINT "SQLCONNECT failed"
```

```
IF verbose THEN CRT "status of ":FUNC:" is ":status

FUNC="SQLALLOCSTMT"
status = SQLALLOCSTMT(connect.env, stmt.env)
IF status NE SQL.SUCCESS THEN PRINT "SQLALLOCSTMT failed"
IF verbose THEN CRT "status of ":FUNC:" is ":status

FUNC="SQLEXECDIRECT"
status = SQLEXECDIRECT(stmt.env, STMT)
IF status NE SQL.SUCCESS THEN PRINT "SQLEXECDIRECT failed"
IF verbose THEN CRT "status of ":FUNC:" is ":status

FUNC="SQLBINDCOL"
status = SQLBINDCOL(stmt.env, 1, SQL.B.NUMBER, PRICE)
IF status NE SQL.SUCCESS THEN PRINT "SQLBINDCOL failed"
IF verbose THEN CRT "status of ":FUNC:" is ":status

retvar = "unknown"
len = 0

CNTR=0
LOOP.LIMIT = 500

LOOP
FUNC="SQLFETCH"
CNTR += 1
status = SQLFETCH(stmt.env)

IF verbose THEN CRT "status of ":FUNC:" is ":status
UNTIL status = SQL.NO.DATA.FOUND

IF status NE SQL.SUCCESS THEN PRINT "SQLFETCH failed"
?PRINT "NAME = ":TRIM(NAME)
IF CNTR > LOOP.LIMIT THEN
STOP "fetch failed: Loop exeeceded the number of cycles allowed"
END

status = SQLGETDATA(stmt.env, 2, SQL.CHAR, retvar, 1024, len)
PRINT "NAME = ":retvar
PRINT "len = ":len
PRINT "Price = ":PRICE
REPEAT

FUNC="SQLFREESTMT"
status = SQLFREESTMT(stmt.env, SQL.DROP)
IF status NE SQL.SUCCESS THEN PRINT "SQLFREESTMT failed, staus =":status
IF verbose THEN CRT "status of ":FUNC:" is ":status

FUNC="SQLDISCONNECT"
status = SQLDISCONNECT(connect.env)
IF status NE SQL.SUCCESS THEN PRINT "SQLDISCONNECT failed"
IF verbose THEN CRT "status of ":FUNC:" is ":status

FUNC="SQLFREECONNECT"
status = SQLFREECONNECT(connect.env)
IF status NE SQL.SUCCESS THEN PRINT "SQLFREECONNECT failed"
IF verbose THEN CRT "status of ":FUNC:" is ":status

FUNC="SQLFREEENV"
status = SQLFREEENV(bci.env)
IF status NE SQL.SUCCESS THEN PRINT "SQLFREEENV failed"
IF verbose THEN CRT "status of ":FUNC:" is ":status
```

```
STOP
```

SQLGetInfo

`SQLGetInfo` returns general information about the ODBC driver and the data source.

This function supports all of the possible requests for information defined in the ODBC 2.0 specification. The *#define*s for *info.type* are contained in the ODBC.H include file.

Syntax

```
status = SQLGetInfo (connect.env, info.type, info.value)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
connect.env	Connection environment.
info.type	The specific information requested. For a list of values, see the following *info.type* tables
info.value	The information returned by `SQLGetInfo`.

ODBC info.type values

The following table lists the valid ODBC values for *info.type*. For more detailed information about information types, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Driver information	
SQL.ACTIVE.CONNECTIONS	SQL.DRIVER.VER
SQL.ACTIVE.STATEMENTS	SQL.FETCH.DIRECTION
SQL.DATA.SOURCE.NAME	SQL.FILE.USAGE
SQL.DRIVER.HDBC	SQL.GETDATA.EXTENSIONS
SQL.DRIVER.HENV	SQL.LOCK.TYPES
SQL.DRIVER.HLIB	SQL.ODBC.API.CONFORMANCE
SQL.DRIVER.HSTMT	SQL.ODBC.SAG.CLI.CONFORMANCE
SQL.DRIVER.NAME	SQL.ODBC.VER
SQL.DRIVER.ODBC.VER	SQL.POS.OPERATIONS
SQL.ROW.UPDATES	SQL.SERVER.NAME
SQL.SEARCH.PATTERN.ESCAPE	
DBMS product information	
SQL.DATABASE.NAME	SQL.DBMS.VER
SQL.DBMS.NAME	
Data source information	
SQL.ACCESSIBLE.PROCEDURES	SQL.OWNER.TERM
SQL.ACCESSIBLE.TABLES	SQL.PROCEDURE.TERM
SQL.BOOKMARK.PERSISTENCE	SQL.QUALIFIER.TERM
SQL.CONCAT.NULL.BEHAVIOR	SQL.SCROLL.CONCURRENCY

SQL.CURSOR.COMMIT.BEHAVIOR	SQL.SCROLL.OPTIONS
SQL.DATA.SOURCE.READ.ONLY	SQL.STATIC.SENSITIVITY
SQL.DEFAULT.TXN.ISOLATION	SQL.TABLE.TERM
SQL.MULT.RESULT.SETS	SQL.TXN.CAPABLE
SQL.MULTIPLE.ACTIVE.TXN	SQL.TXN.ISOLATION.OPTION
SQL.NEED.LONG.DATA.LEN	SQL.USER.NAME
SQL.NULL.COLLATION	
Supported SQL	
SQL.ALTER.TABLE	SQL.ODBC.SQL.OPT.IEF
SQL.COLUMN.ALIAS	SQL.ORDER.BY.COLUMNS.IN.SELECT
SQL.EXPRESSIONS.IN.ORDER.BY	SQL.OWNER.USAGE
SQL.GROUP.BY	SQL.POSITIONED.STATEMENTS
SQL.IDENTIFIER.CASE	SQL.PROCEDURES
SQL.IDENTIFIER.QUOTE.CHAR	SQL.QUALIFIER.LOCATION
SQL.KEYWORDS	SQL.QUALIFIER.NAME.SEPARATOR
SQL.LIKE.ESCAPE.CLAUSE	SQL.QUALIFIER.USAGE
SQL.NON.NULLABLE.COLUMNS	SQL.QUOTED.IDENTIFIER.CASE
SQL.ODBC.SQL.CONFORMANCE	SQL.SPECIAL.CHARACTERS
SQL.SUBQUERIES	SQL.UNION
SQL limits	
SQL.MAX.BINARY.LITERAL.LEN	SQL.MAX.OWNER.NAME.LEN
SQL.MAX.CHAR.LITERAL.LEN	SQL.MAX.PROCEDURE.NAME.LEN
SQL.MAX.COLUMN.NAME.LEN	SQL.MAX.QUALIFIER.NAME.LEN
SQL.MAX.COLUMNS.IN.GROUP.BY	SQL.MAX.ROW.SIZE
SQL.MAX.COLUMNS.IN.ORDER.BY	SQL.MAX.ROW.SIZE.INCLUDES.LONG
SQL.MAX.COLUMNS.IN.INDEX	SQL.MAX.STATEMENT.LEN
SQL.MAX.COLUMNS.IN.SELECT	SQL.MAX.TABLE.NAME.LEN
SQL.MAX.COLUMNS.IN.TABLE	SQL.MAX.TABLES.IN.SELECT
SQL.MAX.CURSOR.NAME.LEN	SQL.MAX.USER.NAME.LEN
SQL.MAX.INDEX.SIZE	
Scalar function information	
SQL.CONVERT.FUNCTIONS	SQL.TIMEDATE.ADD.INTERVALS
SQL.NUMERIC.FUNCTIONS	SQL.TIMEDATE.DIFF.INTERVALS
SQL.STRING.FUNCTIONS	SQL.TIMEDATE.FUNCTIONS
SQL.SYSTEM.FUNCTIONS	
Conversion information	
SQL.CONVERT.BIGINT	SQL.CONVERT.LONGVARCHAR
SQL.CONVERT.BINARY	SQL.CONVERT.NUMERIC
SQL.CONVERT.BIT	SQL.CONVERT.REAL
SQL.CONVERT.CHAR	SQL.CONVERT.SMALLINT
SQL.CONVERT.DATE	SQL.CONVERT.TIME
SQL.CONVERT.DECIMAL	SQL.CONVERT.TIMESTAMP
SQL.CONVERT.DOUBLE	SQL.CONVERT.TINYINT
SQL.CONVERT.FLOAT	SQL.CONVERT.VARBINARY

SQL.CONVERT.INTEGER	SQL.CONVERT.VARCHAR
SQL.CONVERT.LONGVARBINARY	

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLGetTypeInfo

`SQLGetTypeInfo` returns information about an SQL on the data source. You can use `SQLGetTypeInfo` only against ODBC data sources. `SQLGetTypeInfo` returns a standard result set ordered by DATA.TYPE and TYPE.NAME.

Syntax

```
status = SQLGetTypeInfo (statement.env, sql.type)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment.

Parameter	Description
sql.type S	A driver-specific SQL data type, or one of the following: SQL.B.BINARY SQL.BIGINT SQL.BINARY SQL.BIT SQL.C.BINARY SQL.CHAR SQL.DATE SQL.DECIMAL SQL.DOUBLE SQL.FLOAT SQL.INTEGER QL.LONGVARBINARY SQL.LONGVARCHAR SQL.NUMERIC SQL.REAL SQL.SMALLINT SQL.TIME SQL.TIMESTAMP SQL TINYINT SQL.VARBINARY SQL.VARCHAR SQL.WCHAR SQL.WLONGVARCHAR SQL.WVARCHAR

Result set

The following table lists the columns in the result set. For more detailed information about data type information, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide.*

Column name	Data type	Description
TYPE.NAME	Varchar	Data-source-dependent data type name.
DATA.TYPE	Smallint	Driver-dependent or SQL data type.
PRECISION	Integer	Maximum precision of the data type on the data source.
LITERAL.PREFIX	Varchar(128)	Characters used to prefix a literal.
LITERAL.SUFFIX	Varchar(128)	Characters used to terminate a literal.
CREATE.PARAMS	Varchar(128)	Parameters for a data type definition.
NULLABLE	Smallint	Data type accepts null values. Returns one of the following: ▪ SQL.NO.NULLS ▪ SQL.NULLABLE ▪ SQL.NULLABLE.UNKNOWN
CASE.SENSITIVE	Smallint	Character data type is case-sensitive. Returns one of the following: TRUE if data type is a character data type and is case-sensitive FALSE if data type is not a character data type and is not case-sensitive

Column name	Data type	Description
SEARCHABLE	Smallint	How the WHERE clause uses the data type. Returns one of the following: ▪ SQL.UNSEARCHABLE if data type cannot be used ▪ SQL.LIKE.ONLY if data type can be used only with the LIKE predicate ▪ SQL.ALL.EXCEPT.LIKE if data type can be used with all comparison operators except LIKE ▪ SQL.SEARCHABLE if data type can be used with any comparison operator
UNSIGNED.ATTRIBUTE	Smallint	Data type is unsigned. Returns one of the following: TRUE if data type is unsigned FALSE if data type is signed NULL if attribute is not applicable to the data type or the data type is not numeric
MONEY	Smallint	Data type is a money data type. Returns one of the following: TRUE if data type is a money data type FALSE if it is not
AUTO.INCREMENT	Smallint	Data type is autoincrementing. Returns one of the following: TRUE if data type is autoincrementing FALSE if it is not NULL if attribute is not applicable to the data type or the data type is not numeric
LOCAL.TYPE.NAME	Varchar(128)	Localized version of TYPE.NAME.
MINIMUM.SCALE	Smallint	Minimum scale of the data type on the data source.
MAXIMUM.SCALE	Smallint	Maximum scale of the data type on the data source.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLNumParams

`SQLNumParams` returns the number of parameters in an SQL statement.

Use this function after preparing or executing an SQL statement or procedure call to find the number of parameters in an SQL statement. If the statement associated with *statement.env* contains no parameters, *parameters* is set to 0.

Syntax

```
status = SQLNumParams (statement.env, parameters)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment containing the prepared or executed SQL statement.
parameters	Number of parameters in the statement.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLNumResultCols

`SQLNumResultCols` returns the number of columns in a result set.

Use this function after executing an SQL statement to find the number of columns in the result set. If the executed statement was not a SELECT statement or a called procedure that produced a result set, the number of result columns returned is 0.

Use this function when the number of columns to be bound to application variables is unknown, for example, when your program is processing SQL statements entered by users.

Syntax

```
status = SQLNumResultCols (statement.env, cols)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment containing the executed SQL statement.
cols	Number of report columns generated.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLParamOptions

`SQLParamOptions` lets applications load an array of parameter markers in a single `SQLExecDirect` or `SQLExecute` function call. Use this function only when you are connected to an ODBC data source.

Syntax

status = **SQLParamOptions** (*statement.env, option, value*)

`SQLParamOptions` works for all parameter types—output and input/output parameters as well as the more usual input parameters.

Be careful when you use matrices instead of arrays. For example, in the matrix:

dim matrix(10,10)

the elements 1,1, 1,2, ..., 1,10, 2,1, 2,2, ... occupy consecutive memory locations. Since `SQLParamOptions` requires each location specified in `SQLBind` parameter to point to a consecutive series of values in memory, an application using a matrix must load the values of the matrix in the correct order.

When the SQL statement is executed, all variables are checked, data is converted when necessary, and all values in the set are verified to be appropriate and within the bounds of the marker definition. Values are then copied to low-level structures associated with each parameter marker. If a failure occurs while the values are being checked, `SQLExecDirect` or `SQLExecute` returns SQL.ERROR, and *value* contains the number of the row where the failure occurred.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment.
option	One of the following, followed by a value: ▪ SQL.PARAMOPTIONS.SET – *value* is an input variable containing the number of rows to process. It can be an integer from 1 through 1024. ▪ SQL.PARAMOPTIONS.READ – *value* is an output variable containing the number of parameter rows processed by `SQLExecDirect` or `SQLExecute`. As each set of parameters is processed, *value* is updated to the current row number. If `SQLExecDirect` or `SQLExecute` encounters an error, *value* contains the number of the row that failed.

Parameter	Description
value	If *option* is SQL.PARAMOPTIONS.SET, *value* is the number of rows to process. It can be an integer from 1 through 1024. 1 is the default. If *option* is SQL.PARAMOPTIONS.READ, *value* contains the number of parameter rows processed by `SQLExecDirect` or `SQLExecute`. As each set of parameters is processed, *value* is updated to the current row number.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

Example

This example shows how you might use `SQLParamOptions` to load a simple table. Table TAB1 has two columns: an integer column and a CHAR(30) column.

```
$include INCLUDE ODBC.H
arrsize = 20
dim p1(arrsize)
dim p2(arrsize)
SQLINS1 = "INSERT INTO TAB1 VALUES(?,?)"
rowindex = 0

status = SQLAllocEnv(henv)
status = SQLAllocConnect(henv, hdbc)
status = SQLConnect(hdbc, "odbcds", "dbuid", "dbpwd")
status = SQLAllocStmt(hdbc, hstmt)

status = SQLPrepare(hstmt, SQLINS1)
status = SQLBindParameter(hstmt, 1, SQL.B.BASIC, SQL.INTEGER, 0, 0, p1(1),
   SQL.PARAM.INPUT)

status = SQLBindParameter(hstmt, 2, SQL.B.BASIC, SQL.CHAR, 30, 0, p2(1),
   SQL.PARAM.INPUT)

status = SQLParamOptions(hstmt, SQL.PARAMOPTIONS.SET, arrsize)
for index = 1 to arrsize
   p1(index) = index
   p2(index) = "This is row ":index
next index

* now execute, delivering 20 sets of parameters in one network operation

stexec = SQLExecute(hstmt)
status = SQLParamOptions(hstmt, SQL.PARAMOPTIONS.READ, rowindex)

if stexec = SQL.ERROR then
   print "Error in parameter row number ":rowindex
end else
   print rowindex:" parameter marker sets were processed"
```

```
end
```

SQLPrepare

SQLPrepare passes an SQL statement or procedure call to the data source in order to prepare it for execution by SQLExecute.

Use this function to deliver either an SQL statement or a call to an SQL procedure to the data source where it can prepare to execute the passed SQL statement or the procedure. The application subsequently uses SQLExecute to tell the data source to execute the prepared SQL statement or procedure.

What happens when the data source executes the SQLPrepare call depends on the data source. In many cases the data source parses the SQL statement and generates an execution plan that allows rapid, efficient execution of the SQL statement.

Use SQLPrepare and SQLExecute functions when you are issuing SQL statements or calling a procedure repeatedly. You can supply values to a prepared INSERT or UPDATE statement and issue an SQLExecute call each time you change the values of parameter markers. SQLExecute sends the current values of the parameter markers to the data source and executes the prepared SQL statement or procedure with the current values.

Note: Before you issue an SQLExecute call, all parameter markers in the SQL statement or procedure call must be defined using an SQLBindParameter call, otherwise SQLExecute returns an error.

If the parameter type of an SQLBindParameter procedure is SQL.PARAM.OUTPUT or SQL.PARAM.INPUT.OUTPUT, values are returned to the specified program variables.

Syntax

```
status = SQLPrepare (statement.env, statement)
```

ODBC data sources

If you execute a stored procedure or enter a command batch with multiple SELECT statements, the results of only the first SELECT statement are returned.

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment from a previous SQLAllocStmt.
statement	Either an SQL statement or a call to an SQL procedure, to be executed at the data source. To call an SQL procedure, use the following syntax: `[ { ] CALL procedure [ ( [ parameter [ , parameter ] … ] ) ] [ } ]`
procedure	Name of the procedure. If the procedure name contains characters other than alphabetic or numeric, enclose the name in double quotation marks. To embed a single double quotation mark in the procedure name, use two consecutive double quotation marks.

Parameter	Description
parameter	Either a literal value or a parameter marker that marks where to insert values to send to or receive from the data source. Programmatic SQL uses a ? (question mark) as a parameter marker. Use parameters only if the procedure is a subroutine. The number and order of parameters must correspond to the subroutine arguments. For an ODBC data source, parameters should be of the same data type as the procedure requires.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLRowCount

`SQLRowCount` returns the number of rows changed by UPDATE, INSERT, or DELETE statements, or by a called procedure that executes one of these statements.

Statements such as GRANT and CREATE TABLE, which do not update rows in the database, return 0 in *rows*.

For a SELECT statement, a 0 row count is always returned, unless the SELECT statement includes the TO SLIST clause. In that case, `SQLRowCount` returns the number of rows in the select list.

The value of *rows* returned after executing a stored procedure at the data source may not be accurate. It is accurate for a single UPDATE, INSERT, or DELETE statement.

Syntax

```
status = SQLRowCount (statement.env, rows)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment containing the executed SQL statement.
rows	Number of rows affected by the operation.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR

Return value	Description
-2	SQL.INVALID.HANDLE

SQLSetConnectOption

`SQLSetConnectOption` controls some aspects of the connection to a data source.

Syntax

status = **SQLSetConnectOption** (*connect.env, option, value*)

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
connect.env	Connection environment returned from a previous `SQLAllocConnect`.

Options

The following table describes the options available with `SQLSetConnectOption`.

Option	Description
SQL.AUTO.COMMIT	Determines the commit mode for transactions. When you use this option, the connection must already be established, and the SQL.TX.PRIVATE option must be set to SQL.TX.PRIVATE.ON. Valid settings are: ▪ SQL.AUTO.COMMIT.ON – Puts a private connection into autocommit mode. ▪ SQL.AUTO.COMMIT.OFF – Puts a private connection into manual commit mode.
SQL.TX.PRIVATE	Determines if a transaction is controlled by UniBasic or the data source. Valid settings are: ▪ SQL.TX.PRIVATE.ON – Transaction processing is controlled directly by the DBMS on the server. ▪ SQL.TX.PRIVATE.OFF – Transaction processing is fully managed by UniBasic.

Option	Description
SQL.TXN.ISOLATION	Determines the isolation level on the server. When you use this option, the connection must already be established, the SQL.TX.PRIVATE option must be set to SQL.TX.PRIVATE.ON, and no transactions may be active. Valid settings are: ▪ SQL.TXN.READ.UNCOMMITTED – Sets the isolation level on the server to 1. ▪ SQL.TXN.READ.COMMITTED – Sets the isolation level on the server to 2. ▪ SQL.TXN.REPEATABLE.READ – Sets the isolation level on the server to 3. ▪ SQL.TXN.SERIALIZABLE – Sets the isolation level on the server to 4.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

Private transactions

SQL.TX.PRIVATE.ON frees the connection from being managed by the UniData transaction manager. When you make a connection private, the application can use the SQL.AUTOCOMMIT option to put the connection into either autocommit mode or manual commit mode. By default, private connections are in autocommit mode, in which each SQL statement is treated as a separate transaction, committed after the statement is executed.

In manual commit mode the application can do either of the following:

- Use the `SQLTransact` function to commit or roll back changes to the database.
- Set the SQL.AUTOCOMMIT option of `SQLSetConnectOption` to SQL.AUTOCOMMIT.ON. This commits any outstanding transactions and returns the connection to autocommit mode.

You must return the connection to autocommit mode before using `SQLDisconnect` to close the connection. You can do this in two ways:

- Set the SQL.AUTOCOMMIT option of `SQLSetConnectOption` to SQL.AUTOCOMMIT.ON
- Set the SQL.TX.PRIVATE option of `SQLSetConnectOption` to SQL.TX.PRIVATE.OFF

When a connection is private, SQL.TXN.ISOLATION lets the application define the default transaction isolation level at which to execute server operations. To determine what isolation levels the data source supports, use the SQL.TXN.ISOLATION.OPTION option of the `SQLGetInfo` function. This returns a bitmap of the options the data source supports. The application can then use the UniBasic BIT functions to determine whether a particular bit is set in the bitmap.

Use `SQLSetConnectOption` with the SQL.TXN.ISOLATION option only in the following two places:

- Immediately following an `SQLConnect` function call
- Immediately following an `SQLTransact` call to commit or roll back an operation

Whenever you execute an SQL statement, a new transaction exists, which makes setting the SQL.TXN.ISOLATION option illegal. If a transaction is active when the SQL.TXN.ISOLATION.OPTION is set, UniData BCI returns SQL.ERROR and sets SQLSTATE to S1C00.

SQLSetParam

`SQLSetParam` is a synonym for `SQLBindParameter`.

SQLSpecialColumns

`SQLSpecialColumns` gets information about columns in a table.

`SQLSpecialColumns` lets applications scroll forward and backward in a result set to get the most recent data from a set of rows. Columns returned for column type SQL.BEST.ROWID are guaranteed not to change while positioned on that row. Columns of the row ID can remain valid even when the cursor is not positioned on the row. The application can determine this by checking the SCOPE column in the result set.

Once the application gets values for SQL.BEST.ROWID, it can use these values to reselect that row within the defined scope. The SELECT statement is guaranteed to return either no rows or one row.

Columns returned for SQL.BEST.ROWID can always be used in an SQL select expression or WHERE clause. However, `SQLColumns` does not necessarily return these columns. If no columns uniquely identify each row in the table, `SQLSpecialColumns` returns a row set with no rows; a subsequent call to `SQLFetch` returns SQL.NO.DATA.FOUND.

Columns returned for column type SQL.ROWVER let applications check if any columns in a given row have been updated while the row was reselected using the row ID.

If *col.type*, *IDscope*, or *null* specifies characteristics not supported by the data source, `SQLSpecialColumns` returns a result set with no rows, and a subsequent call to `SQLFetch` returns SQL.NO.DATA.FOUND.

For complete details about the `SQLSpecialColumns` function, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Syntax

```
status = SQLSpecialColumns (statement.env, col.type, schema, owner,
tablename, IDscope, null)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment.
col.type	Type of column to return. *col.type* is one of the following: ▪ SQL.BEST.ROWID – Returns the best column or set of columns that uniquely identifies a row in a table. ▪ SQL.ROWVER – Returns the column or columns that are automatically updated when any value in the row is updated by a transaction.

Parameter	Description
schema	Qualifier name for *tablename*. If a driver supports qualifiers for some tables but not others, use an empty string for tables that do not have qualifiers.
owner	Name of the owner of the table. If a driver supports owners for some table but not others, use an empty string for tables that do not have owners.
tablename	Name of the table.
IDscope	Minimum required scope of the row ID. *IDscope* is one of the following: ▪ SQL.SCOPE.CURROW – Row ID is guaranteed to be valid only while positioned on that row. ▪ SQL.SCOPE.TRANSACTION – Row ID is guaranteed to be valid for the duration of the current transaction. ▪ SQL.SCOPE.SESSION – Row ID is guaranteed to be valid for the duration of the session.
null	Can be one of the following: ▪ SQL.NO.NULLS – Excludes special columns that can have null values. ▪ SQL.NULLABLE – Returns special columns even if they can have null values.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLStatistics

`SQLStatistics` gets a list of statistics about a single table and its indexes. Use this function only when you are connected to an ODBC data source.

`SQLStatistics` returns information as a standard result set ordered by NON.UNIQUE, TYPE, INDEX.QUALIFIER, INDEX.NAME, and SEQ.IN.INDEX. The result set combines statistics for the table with statistics for each index.

`SQLStatistics` might not return all indexes. For example, a driver might return only the indexes in files in the current directory. Applications can use any valid index regardless of whether it is returned by `SQLStatistics`.

Syntax

```
status = SQLStatistics (statement.env, schema, owner, tablename,
index.type, accuracy)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment.
schema	Qualifier name for *tablename*. If a driver supports qualifiers for some tables but not others, use an empty string for tables that do not have qualifiers.
owner	Name of the owner of the table. If a driver supports owners for some table but not others, use an empty string for tables that do not have owners.
tablename	Name of the table.
index.type	One of the following: ▪ SQL.INDEX.UNIQUE ▪ SQL.INDEX.ALL
accuracy	The importance of the CARDINALITY and PAGES columns in the result set: ▪ SQL.ENSURE – The driver unconditionally gets the statistics. ▪ SQL.QUICK – The driver gets results only if they are readily available from the server. The driver does not ensure that the values are current.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

The lengths of VARCHAR columns are maximums; the actual lengths depend on the data source. Use `SQLGetInfo` to determine the actual lengths of the TABLE.QUALIFIER, TABLE.OWNER, TABLE.NAME, and COLUMN.NAME columns.

Column name	Data type	Description
TABLE.QUALIFIER	Varchar(128)	Table qualifier identifier (schema) of the table. The null value is returned if it is not applicable to the data source. If a driver supports qualifiers for some tables but not others, it returns an empty string for tables without qualifiers.
TABLE.OWNER	Varchar(128)	Name of the owner of the table. The null value is returned if it is not applicable to the data source. If a driver supports owners for some tables but not others, it returns an empty string for tables without owners.
TABLE.NAME	Varchar(128) Not null	Name of the table.

Column name	Data type	Description
NON.UNIQUE	Smallint	The index prohibits duplicate values: TRUE if the index values can be non-unique. FALSE if the index values must be unique. NULL if TYPE is SQL.TABLE.STAT.
INDEX.QUALIFIER	Varchar(128)	Index qualifier identifier used by the DROP INDEX statement. The null value is returned if the data source does not support index qualifiers or if TYPE is SQL.TABLE.STAT. If a nonnull value is returned, it must be used to qualify the index name in a DROP INDEX statement, otherwise the TABLE.OWNER name should be used to qualify the index name.
INDEX.NAME	Varchar(128)	Name of the index. The null value is returned if TYPE is SQL.TABLE.STAT.
TYPE	Smallint Not null	Type of information returned: ▪ SQL.TABLE.STAT indicates a statistic for the table. ▪ SQL.INDEX.CLUSTERED indicates a clustered index. ▪ SQL.INDEX.HASHED indicates a hashed index. ▪ SQL.INDEX.OTHER indicates another type of index.
SEQ.IN.INDEX	Smallint	Column sequence number in index, starting with 1. The null value is returned if TYPE is SQL.TABLE.STAT.
COLUMN.NAME	Varchar(128)	Name of a column. If the column is based on an expression, the expression is returned. If the expression cannot be determined, an empty string is returned. If the index is filtered, each column in the filter condition is returned (this may require more than one row). The null value is returned if TYPE is SQL.TABLE.STAT.
COLLATION	Char(1)	Sort sequence for the column: ▪ A indicates ascending. ▪ B indicates descending. The null value is returned if the data source does not support column sort sequence.
CARDINALITY	Integer	Number of rows in the table if TYPE is SQL.TABLE.STAT. Number of unique values in the index if TYPE is not SQL.TABLE.STAT. The null value is returned if the value is not available from the data source or if it is not applicable to the data source.

Column name	Data type	Description
PAGES	Integer	Number of pages for the table if TYPE is SQL.TABLE.STAT. Number of pages for the index if TYPE is not SQL.TABLE.STAT. The null value is returned if the value is not available from the data source or if it is not applicable to the data source.
FILTER.CONDITION	Varchar(128)	If the index is filtered, the filter condition, or an empty string if the filter condition cannot be determined. The null value is returned if the index is not filtered, or if it cannot be determined that the index is filtered, or TYPE is SQL.TABLE.STAT.

If the row in the result set corresponds to a table, the driver sets TYPE to SQL.TABLE.STAT and sets the following columns to NULL:

- NON.UNIQUE
- INDEX.QUALIFIER
- INDEX.NAME
- SEQ.IN.INDEX
- COLUMN.NAME
- COLLATION

If CARDINALITY or PAGES are not available from the data source, the driver sets them to NULL.

For complete details about the `SQLStatistics` function, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

SQLTables

`SQLTables` returns a result set listing the tables matching the search patterns. Use this function only when you are connected to an ODBC data source.

This function returns *statement.env* as a standard result set of five columns containing the schemas, owners, names, types, and remarks for all tables found by the search. The search criteria arguments can contain a literal, an SQL LIKE pattern, or be empty. If a literal or LIKE pattern is specified, only values matching the pattern are returned. If a criterion is empty, tables with any value for that attribute are returned. *owner* cannot specify a LIKE pattern.

Syntax

```
status = SQLTables (statement.env, schema, owner, tablename, type)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
statement.env	SQL statement environment.
schema	Schema name search pattern.
owner	Table owner number search pattern.

Parameter	Description
tablename	Table name search pattern.
type	Table type (one of the following: BASE TABLE, VIEW, ASSOCIATION, or TABLE) search pattern.

You can access the result set with `SQLFetch`. These five columns have the following descriptors:

Column name	Data Type
TABLE.SCHEMA	VARCHAR(128)
TABLE.OWNER	VARCHAR(128)
TABLE.NAME	VARCHAR(128)
TABLE.TYPE	VARCHAR(128)
REMARKS	VARCHAR(254)

Special search criteria

Three special search criteria combinations enable an application to enumerate the set of schemas, owners, and tables:

Table qualifier	Table owner	Table name	Table type	Return is..
%	empty string	empty string	*ignored*	Set of distinct schema names
empty string		empty string	*ignored*	Set of distinct table owners
empty string	empty string	empty string	%	Set of distinct table types

The ability to obtain information about tables does not imply that you have any privileges on those tables.

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLSTATE values

The following table describes the SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1008	Cancelled. Execution of the statement was stopped by an `SQLCancel` call.
S1010	Function sequence error. The *statement.env* specified is currently executing an SQL statement.

SQLSTATE	Description
S1C00	The table owner field was not numeric.
24000	Invalid cursor state. Results are still pending from the previous SQL statement. Use `SQLCancel` to clear the results.
42000	Syntax error or access violation. This can be caused by a variety of reasons. The native error code returned by the `SQLError` call indicates the specific error that occurred.

SQLTransact

`SQLTransact` requests a COMMIT or ROLLBACK for all SQL statements associated with a connection or all connections associated with an environment. Use this function only when you are connected to an ODBC data source.

This function provides the same transaction functions as the UniBasic statement TRANSACTION COMMIT. When *connect.env* is a valid connection environment with SQL.AUTOCOMMIT set to OFF, `SQLTransact` commits the connection.

To use `SQLTransact`, all of the following conditions must be met:

- The SQL.TX.PRIVATE option of `SQLSetConnectOption` is set to SQL.TX.PRIVATE.ON.
- The SQL.AUTOCOMMIT option is set to SQL.AUTOCOMMIT.OFF.
- The connection is active.

Setting *bci.env* to a valid environment handle and *connect.env* to SQL.NULL.HDBC requests the server to try to execute the requested action on all *hdbc*s that are in a connected state.

If any action fails, SQL.ERROR is returned, and the user can determine which connections failed by calling `SQLError` for each connection environment in turn.

If you call `SQLTransact` with a *type* of SQL.COMMIT or SQL.ROLLBACK when no transaction is active, SQL.SUCCESS is returned.

Syntax

```
status = SQLTransact (bci.env, connect.env, type)
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
bci.env	UniData BCI environment.
connect.env	Connection environment or SQL.NULL.HDBC.
type	One of the following: ▪ SQL.COMMIT – Writes all modified data to the data source, releases all lock acquired by the current transaction, and terminates the transaction. ▪ SQL.ROLLBACK – Discards any changes written during the transaction and terminates it .

Return values

The following table describes the return values of this function.

Return value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLSTATE values

The following table describes the SQLSTATE values for the `SQLTransact` function.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1012	*type* did not contain SQL.COMMIT or SQL.ROLLBACK.
08003	No connection is active on *connect.env*.
08007	The connection associated with the transaction failed during the execution of the function. It cannot be determined if the requested operation completed before the failure.

Appendix A: Data conversion

This appendix describes the UniBasic and SQL data types you can specify with the `SQLBindParameter` and `SQLBindCol` calls. It also explains how data is converted.

You use `SQLBindParameter` to convert UniBasic data to an SQL data type that can be sent to the data source. You specify the UniBasic data type as SQL.B.BASIC.

You use `SQLBindCol` to convert SQL data, retrieved from the data source, to a UniBasic data type. You can specify that data retrieved from the data source be stored locally as:

- A character string (SQL.B.CHAR)
- A bit string (SQL.B.BINARY)
- A number (SQL.B.NUMBER)
- An internal date (SQL.B.INTDATE)
- An internal time (SQL.B.INTTIME)

You can also specify SQL.B.DEFAULT. This lets the SQL data type determine the UniBasic data type to which to convert data from the data source. The NULL data type requires no conversion.

The following table shows UniData BCI data types and their UniBasic counterparts.

SQL client Interface data type	BASIC data type	Used in calls
SQL.B.CHAR	Character string.	`SQLBindCol`
SQL.B.BINARY	Bit string (raw data)	`SQLBindCol` `SQLBindParameter`
SQL.B.NUMBER	Integer. Double.	`SQLBindCol`
SQL.B.DEFAULT	SQL data type at the source determines how to store locally.	`SQLBindCol`
SQL.B.BASIC	The data determines the BASIC data type. The data must be string, integer, or double.	`SQLBindParameter`

The next table shows the SQL data types and their UniData BCI definitions. These SQL data types can be used when you are connected to UniData data sources and to ODBC data sources.

SQL data type	SQL client interface definition	Description and SQL client interface defaults
SQL.CHAR	CHAR (*n*)	Fixed-length character string precision = *n* where 1 <= *n* <= 254
SQL.VARCHAR SQL.LONGVARCHAR	VARCHAR (*n*)	Variable-length character string precision = *n* where 1 <= *n* <= 65535
SQL.WCHAR	NCHAR (*n*)	Fixed-length national character string precision = *n* where 1 <= *n* <= 254
SQL.WVARCHAR SQL.WLONGVARCHAR	NVARCHAR (*n*)	Variable-length national character string precision = *n* where 1 <= *n* <= 65535
SQL.BINARY	BIT (*n*)	Bit string (raw data)

SQL data type	SQL client interface definition	Description and SQL client interface defaults
SQL.VARBINARY SQL.LONGVARBINARY	VARBIT (*n*)	Bit string (raw data)
SQL.DECIMAL	DECIMAL (*p*,*s*)	Signed exact numeric precision = *p*, scale = *s*
SQL.NUMERIC	NUMERIC (*p*,*s*)	Signed exact numeric precision = *p*, scale = *s*
SQL.SMALLINT	SMALLINT	Signed exact numeric precision = 5, scale = 0
SQL.INTEGER	INTEGER	Signed exact numeric precision = 10, scale = 0
SQL.REAL	REAL	Signed approximate numeric precision = 7
SQL.FLOAT	FLOAT	Signed approximate numeric precision = 15
SQL.DOUBLE	DOUBLE PRECISION	Signed approximate numeric precision = 15
SQL.DATE	DATE	Data source dependent
SQL.TIME	TIME	Internal time

This table represents the SQL data types currently supported by UniData BCI. Some data types encountered on ODBC data sources cannot be mapped into any of these types. These data types typically represent binary bit or byte streams. Your programs cannot fetch data from such column types. If you try to execute an SQL `SELECT` statement that produces a column with an unsupported data type, an application error (SQLSTATE code S1004) is returned to the execution call. When this happens, use the `SQLDescribeCol` or `SQLColAttributes` function to determine which column you requested contains an unsupported binary data type. The SQL data type for these columns is returned as SQL.BINARY.

The following table lists SQL data types you can use when connected to certain ODBC data sources.

SQL data type	Description
SQL.LONGVARCHAR	Text columns that may be longer than the limits imposed by CHAR or VARCHAR data types.
SQL.TIMESTAMP	Date and time information in the formats *yyyy-mm-dd* and *hh*:*mm*:*ss*. ODBC data sources vary in compliance with the ODBC specification regarding what to do with a missing component. For safety, always supply both components of a timestamp parameter.
SQL.BIGINT	Signed integer up to 19 digits, unsigned integer up to 20 digits. Valid only for data sources, such as Oracle, that support integers longer than 10 digits.
SQL.TINYINT	Signed integer from 0 through 255; unsigned integer from -128 through +127. Use the `SQLGetTypeInfo` function to determine if the column is signed or unsigned

SQL data type	Description
SQL.BIT	Data values of 0 or 1.

Converting UniBasic data to SQL data

Use the `SQLBindParameter` call to specify the SQL data type to which to convert outgoing UniBasic data supplied for a parameter marker. This section describes how UniData BCI converts outgoing data.

`SQLBindParameter` verifies that the UniBasic data type is either SQL.B.BASIC, SQL.B.BINARY, SQL.B.INTDATE, or SQL.B.INTTIME. It does not check whether the data type is consistent with a data conversion that may occur later.

Both `SQLExecute` and `SQLExecDirect` calls check the data type and convert the data. Data from locations indicated in `SQLBindParameter` must be one of the following:

- A number (integer or double data types)
- A character string
- A bit string (raw data)
- A subroutine
- A null value

Any other kind of data returns an error to the `SQLExecute` or `SQLExecDirect` call.

When converting UniBasic character string data to numeric SQL data types, all numbers are rounded to 15 digits. The `SQLSetConnectOption` flag, SQL.TRUNC.ROUND, is ignored.

Precision and scale

For ODBC data sources, precision is observed for the following SQL data types:

- SQL.CHAR
- SQL.VARCHAR
- SQL.LONGVARCHAR
- SQL.WCHAR
- SQL.WVARCHAR
- SQL.W.LONGVARCHAR
- SQL.BINARY
- SQL.VARBINARY
- SQL.LONGVARBINARY
- SQL.NUMERIC
- SQL.DECIMAL

Scale is observed for the following data types:

- SQL.DECIMAL
- SQL.NUMERIC

All data sent to ODBC data sources is checked by the ODBC driver and database engine. Thus, conversion failures can be detected either by UniData BCI, by the ODBC driver, or by the underlying DBMS.

UniBasic to SQL CHAR, VARCHAR, LONGVARCHAR, WCHAR, WVARCHAR, or WLONGVARCHAR

No data conversion is necessary when converting UniBasic character string data to these data types. When the data is a UniBasic subroutine, the subroutine name is used as the string value.

If the string is longer than the precision of the column as specified in the `SQLBindParameter` or `SQLSetParam` call, UniData BCI returns SQL.ERROR, sets SQLSTATE to 01004. Other errors can be returned from the data source if the string exceeds the width of the column as defined in the data column.

UniBasic to SQL.BINARY, SQL.VARBINARY, or SQL.LONGVARBINARY

No data conversion is done when converting BASIC bit string data to SQL binary data types.

If the string is longer than the precision of the column as specified in the `SQLBindParameter` or `SQLSetParam` call, UniData BCI returns SQL.ERROR, sets SQLSTATE to 01004. Other errors can be returned from the data source if the string exceeds the width of the column as defined in the data column.

UniBasic to SQL DECIMAL and NUMERIC

If a UniBasic string contains invalid nonnumeric characters, UniData BCI returns SQL.ERROR and sets SQLSTATE to 22005.

If the precision of the string and scale are consistent with the arguments of the `SQLBindParameter` or `SQLSetParam` call, the string is sent to the ODBC driver for further checks.

If the number of significant digits (digits to the left of the decimal point) is greater than the specified precision, SQL.ERROR is returned with an SQLSTATE of 22003.

If the number of insignificant digits (digits to the right of the decimal point) is greater than the defined scale of the column, SQL.SUCCESS.WITH.INFO is returned with an SQLSTATE of 01004, indicated that fractional truncation has occurred.

UniBasic to SQL INTEGER, SMALLINT, TINYINT, BIGINT, and BIT

If a UniBasic string is nonnumeric, UniData BCI returns SQL.ERROR and sets SQLSTATE to 22005.

If the string represents a number that falls outside the limits of the SQL data type of the column, SQL.ERROR is returned with an SQLSTATE of 22003.

Some data sources such as SYBASE treat the TINYINT data type as unsigned, others treat it as signed. Use `SQLGetTypeInfo` to determine whether the type is signed or unsigned.

For SQL.BIT binding, the application should use only the values 0 and 1. ODBC drivers vary as to how they handle other values. Some may return SQL.ERROR, others may return SQL.SUCCESS.WITH.INFO and deliver the integer part of a fractional number.

UniBasic to SQL REAL, FLOAT, and DOUBLE

If a UniBasic string is nonnumeric, UniData BCI returns SQL.ERROR and sets SQLSTATE to 22005.

These data types are passed directly to the ODBC driver and the data source for handling. Drivers and data sources vary as to how they handle numbers trying to represent too many digits in the number. Generally data sources represent 15 digits of precision. Digits beyond 15 may be lost, perhaps silently.

UniBasic to SQL DATE

The `SQLBindParameter` call can contain SQL.B.BASIC or SQL.B.INTDATE as the UniBasic data type. If a UniBasic string is a date in external format, use SQL.B.BASIC. If the string is a date in internal format, use SQL.B.INTDATE.

If you specify a UniBasic data type of SQL.B.INTDATE and the SQL data type is not SQL.DATE, UniData BCI returns an error and sets SQLSTATE to 07006. If the date is invalid, UniData BCI returns SQL.ERROR and sets SQLSTATE to 22008.

Dates are sent to ODBC data sources in the following format:

yyyy-mm-dd

Dates already in external format (SQL.B.BASIC) need no conversion. UniData BCI accepts dates in any valid UniData external format and converts them to the correct format for the ODBC driver.

UniBasic to SQL TIME

The `SQLBindParameter` call can contain SQL.B.BASIC or SQL.B.INTTIME as the UniBasic data type. If the string is a time in external format (*hh*:*mm*:*ss*), use SQL.B.BASIC. If the string is a time in internal format, use SQL.B.INTTIME.

All times in external format (SQL.B.BASIC) must be specified using the following format:

[*h*] *h*: [*m*] *m*: [*s*] *s*

If the hour value is greater than 24, the value is divided by 24 and the remainder is used for the hour. So 50:01:02 is equivalent to 02:01:02. The minutes and seconds values must be from 0 to 59; if they are not, SQL.ERROR is returned and SQLSTATE is set to 22008.

If you specify a UniBasic data type of SQL.B.INTTIME, the value is interpreted as the number of seconds since midnight. The value should be from 0 (midnight) to 86399. On ODBC data sources, if a value is outside this range, UniData BCI returns SQL.ERROR and sets SQLSTATE to 22008.

UniBasic to SQL TIMESTAMP

The TIMESTAMP data type is available only for ODBC data sources.

The `SQLBindParameter` call should contain SQL.B.BASIC as the UniBasic data type and SQL.TIMESTAMP as the SQL data type. The format for timestamp data is as follows:

yyyy-mm-ddhh :*mm*:*ss*

Be sure to supply both parts of this string, because ODBC drivers vary in how they handle a timestamp lacking a date or a time.

If either the date or the time is invalid, SQL.ERROR is returned and SQLSTATE is set to 22008.

Converting SQL data to UniBasic data

Use the `SQLBindCol` call to specify the BASIC data type to which to convert incoming SQL data. This section describes how UniData BCI converts incoming data.

You can specify four UniBasic data types:

- SQL.B.CHAR – Converts to character string data
- SQL.B.BINARY – Converts to bit string (raw) data
- SQL.B.NUMBER – Converts to integer or double
- SQL.B.DEFAULT – Uses the SQL data type to determine how to convert

Unlike `SQLBindParameter`, which does not send truncated data to the data source, the `SQLFetch` function associated with `SQLBindCol` delivers rounded or truncated data to the UniBasic program.

Conversions using the SQL.B.DEFAULT data type follow these rules:

- For the character string and bit string SQL data types, SQL.B.DEFAULT is equivalent to using SQL.B.CHAR.
- For the numeric SQL data types, SQL.B.DEFAULT is equivalent to specifying SQL.B.NUMBER at the `SQLBindCol` call. Data is stored in either integer or double form.

Converting SQL character types to UniBasic data types

There are six SQL character data types:

- SQL.CHAR
- SQL.VARCHAR
- SQL.LONGVARCHAR
- SQL.WCHAR
- SQL.WVARCHAR
- SQL.WLONGVARCHAR

SQL character data types to SQL.B.CHAR and SQL.B.DEFAULT

Data does not need to be converted. Space is allocated and the string is stored in the UniData variable. Trailing spaces are deleted. Multivalued data is passed through with value marks.

SQL character data types to SQL.B.NUMBER

Nonnumeric SQL data returns SQL.ERROR and sets SQLSTATE to 22005.

Numeric SQL data is rounded to 15 digits. If the number of significant digits (excluding trailing zeros) exceeds 15, UniData BCI returns SQL.ERROR and sets SQLSTATE to 01004. If there is a fractional part and the number of digits without trailing zeros exceeds 15, SQLSTATE is set to 22001. Otherwise SQL.SUCCESS is returned.

SQL data	BASIC result	SQLSTATE
JONES	----	22005
123456789	123456789	00000
1234567890123456	----	01004
123456789012345	123456789012345	00000
1234567890123450000	1234567890123450000	00000
1.2e18	1200000000000000000	00000
123e-11	0.00000000123	00000
1234.567890123456789	1234.56789012346	22001
123456789012345.1	123456789012345	22001
12345678901234.1	12345678901234.1	00000
12345678901234.6	12345678901234.6	00000
12345678901234.567	12345678901234.6	22001

Converting SQL binary types to UniBasic data types

There are three SQL binary data types:

- SQL.BINARY
- SQL.VARBINARY
- SQL.LONGVARBINARY

SQL binary data types to SQL.B.BINARY and SQL.B.DEFAULT

Raw data is not converted. Space is allocated and the string is stored in the UniBasic variable.

Converting SQL numeric types to UniBasic data types

There are seven SQL numeric data types:

- SQL.DECIMAL
- SQL.NUMERIC
- SQL.SMALLINT
- SQL.INTEGER
- SQL.REAL
- SQL.FLOAT
- SQL.DOUBLE

SQL numeric types to SQL.B.CHAR

The number is put in the UniBasic variable in ASCII format.

SQL numeric types to SQL.B.NUMBER and SQL.B.DEFAULT

SMALLINT and INTEGER types are stored as UniBasic integers. All others are stored as doubles.

SQL data	SQL type	BASIC result	SQLSTATE
12345	SMALLINT	12345	00000
123456789	INTEGER	123456789	00000
123456789.	FLOAT	123456789.	00000
12345678901234567.25	DOUBLE	12345678901234600	00000
1234.37218738172312	DOUBLE	1234.3721873817	00000

Converting SQL date, time, and timestamp types to UniBasic types

UniData BCI returns an SQL date or time a character string.

You cannot convert any of these SQL data types to SQL.B.NUMBER. If you try, `SQLFetch` generates SQL.ERROR and sets SQLSTATE to 07006.

SQL DATE data to SQL.B.INTDATE

UniData BCI accepts dates in external format and converts them to dates in internal format.

SQL DATE and TIME data to SQL.B.CHAR and SQL.B.DEFAULT

Dates from ODBC data sources are returned in the following format:

yyyy-mm-dd

Times from ODBC data sources are returned in the following format:

hh:*mm*:*ss*

If the date or time is not valid, `SQLFetch` returns SQL.ERROR and sets SQLSTATE to 22008.

SQL TIME data to SQL.B.INTTIME

Times from ODBC data sources are converted to UniData times in internal format via the MTS conversion code. The resulting integer is the data returned in the bound variable.

SQL TIMESTAMP data to SQL.B.CHAR and SQL.B.DEFAULT

The TIMESTAMP data type is available from ODBC data sources. Timestamp data is returned in the following formats:

yyyy-mm-dd

hh:*mm*:*ss*

SQL TIMESTAMP data to SQL.B.INTDATE and SQL.B.INTTIME

The date part of a TIMESTAMP value is converted to a UniData date in internal format via a D2 conversion code. The time part of a TIMESTAMP value is converted to a UniData time in internal format via an MTS conversion code.

The resulting integer is the data returned in the bound variable.

Appendix B: UniData BCI demo program

This appendix describes a demonstration program that shows how to use UniData BCI. The program does the following:

- Gathers information to log on to a data source
- Connects to the data source
- Drops and creates the tables on the data source
- Reads the UniData file and inserts the data into the data source table
- Selects the file from the data source and displays it on the screen

The demonstration program is called BCI.DEMO. It is in the BP file of the UniData demo account. For information about how to run the BCI.DEMO program, see Getting started, on page 10.

Main program

First the program includes the UniData BCI definitions from the ODBC.H file:

```
* Include the ODBC definitions
$INCLUDE INCLUDE ODBC.H
```

The next section gathers the name of the data source, the user name and password for the server operating system, and information for the data source. The DBMS information is often a user name and a password.

```
.
.
SET.CONNECT.INFO:
027: *
028: * ------------------------------------------------------------------------
-
029: * datasource   == Name of the established ODBC datasource
030: * username         == Name of user as defined on the SQL Database Engine
031: * passwd           == Password of user
032: * ========================================================================
=
033: *
034: datasource = "Server7"
035: username = "sa"
036: passwd = ""
037: RETURN
```

The next section of the program establishes the BCI environment. First, the program allocates the database environment, as shown in the following example:

```
* Step 1 - Allocate Database Environment
* ========================================================================

*
STATUS = SQLAllocEnv(database.env)
*
MODULE = "ESTABLISH.BCI"
ENVTYPE = "Database"
Fn = "SQLAllocEnv"
GOSUB CHECK.STATUS
```

```
*
* ------------------------------------------------------------------------
```

Next, the program allocated the connection environment, as shown in the following example:

```
* Step 2 - Allocate Connection Environment
* ========================================================================

*
STATUS = SQLAllocConnect(database.env,connection.env)
*
ENVTYPE = "Connection"
fn = "SQLAllocConnect"
GOSUB CHECK.STATUS
*
CRT "Attempting connect to ":datasource:" with user id ":username
*
* ------------------------------------------------------------------------
```

Next, the program connects to the database.

```
* Step 3 - Connecting to Database
* ========================================================================
*
STATUS = SQLConnect(connection.env, datasource, username, passwd)
*
ENVTYPE = "Connection"
GOSUB CHECK.STATUS
RETURN
*
ALLOC.STATEMENT.ENV:
*
* ------------------------------------------------------------------------
```

The final step in setting up the BCI environment is establishes the SQL statement environment, used when executing SQL statement functions.

```
* Step 4 - Allocate Statement Environment
*               The Statement Environment is used when executing SQL
*               statement functions
* ========================================================================
*
STATUS = SQLAllocStmt(connection.env,statement.env)
*
MODULE = "ALLOC.statement.env"
Fn = "SQLAllocStmt"
ENVTYPE = "Connection"
GOSUB CHECK.STATUS
RETURN
*
```

After making the connection, the program creates some local UniData files, creates dictionaries, and populates them with data:

```
*
DIM DICT(8)
f = @FM
DICT(2) = "ID": f:"D":f:0:f:f:f:"10L":f:"S":f:f:"CHARACTER,10":f
DICT(3) = "NAME": f:"D":f:1:f:f:f:"10L":f:"S":f:f:"CHARACTER,10":f
DICT(4) = "GRADE":f:"D":f:2:f:"MD0":f:f:"10R":f:"S":f:f:"INTEGER":f
DICT(5) = "CITY": f:"D":f:3:f:f:f:"15L":f:"S":f:f:"CHARACTER,15":f
```

```
DICT(6) = "@REVISE": f: "PH":f:f:f:f:f:f:f:f
DICT(7) = "@":f:"PH":f:"ID.SUP ID NAME GRADE CITY":f:f:f:f:f:f:f
DICT(8) = "@KEY":f:"PH":f:"ID":f:f:f:f:f:f:f
*
CRT "Deleting local EMPFILE file"
EXECUTE "DELETE.FILE EMPFILE FORCE"
EXECUTE "CREATE.FILE EMPFILE 3"
*
OPEN "DICT","EMPFILE" TO D.EMPFILE ELSE STOP "Failed to open DICT EMPFILE
REC = ""
FOR INDEX = 2 TO 8
        ID = DICT(INDEX)<1>
        FOR I = 2 TO 9
            REC<I-1> = DICT(INDEX)<I>
        NEXT I
        WRITE REC TO D.EMPFILE, ID
NEXT INDEX
RETURN
*
LOAD.LOCAL.FILE:
*
* ------------------------------------------------------------------------
* This subroutine loads data into the local UniData file
* ========================================================================
*
DIM EMPDATA(5)
EMPDATA(1) = "E1":@FM:"Alice":@FM: 12:@FM:"Deale"
EMPDATA(2) = "E2":@FM:"Betty":@FM: 10:@FM:"Vienna"
EMPDATA(3) = "E3":@FM:"Carmen":@FM: 13:@FM:"Vienna"
EMPDATA(4) = "E4":@FM:"Don":@FM: 12:@FM:"Deale"
EMPDATA(5) = "E5":@FM:"Ed":@FM: 13:@FM:"Akron"
*
* ------------------------------------------------------------------------
* Clear the files and then load them up
* ========================================================================
*
EXECUTE "CLEAR.FILE EMPFILE"
OPEN "EMPFILE" TO EMPFILE ELSE STOP "Failed to open EMPFILE File"
FOR INDEX = 1 TO 5
REC = EMPDATA(INDEX)
ID = REC<1>
DREC = REC<2>:@FM:REC<3>:@FM:REC<4>
WRITE DREC TO EMPFILE, ID
NEXT INDEX
RETURN
*
```

The program now creates a UniBasic CreateStmt variable. The data types depend on the type of data you are using, and the data types available on the data source. You can also use the `SQLGetTypeInfo` function to retrieve supported data types. This example uses the CHAR data type.

```
CreateStmt = "CREATE TABLE EMPTABLE( ID CHAR(10), NAME CHAR(10), GRADE CHAR
CITY CHAR(15))"
```

Now the program creates the DropStmt variable to drop or remove EMPTABLE.

```
DropStmt = "DROP TABLE EMPTABLE"
*
CRT "Dropping EMPTABLE table in ":datasource
```

The program now executes the DropStmt and the CreateStmt through the statement environment.

```
*The drop statement is executed through the Statement Environment
* ========================================================================

*
STATUS = SQLExecDirect(statement.env, DropStmt)
*
CRT "Creating EMPTABLE table in ":datasource
*

* ------------------------------------------------------------------------
* The create statement executed via the Statement Environment
* ========================================================================

*
STATUS = SQLExecDirect(statement.env, CreateStmt)
*
MODULE = "CREATE.TABLE"
Expect = ""
ENVTYPE = "Statement"
GOSUB CHECK.STATUS
RETURN
*
```

The program now creates the InsertStmt UniBasic variable, using place holders or parameter markers to mark where the data will be loaded.

```
InsertStmt = "INSERT INTO EMPTABLE VALUES (?, ?, ?, ?)"
*
```

SQLPrepare now passes the InsertStmt to the data source. The database usually parses the statement in preparation for the execute statement.

```
*
STATUS = SQLPrepare(statement.env, InsertStmt)
*
MODULE = "LOAD.TABLE"
Fn = "SQLPrepare"
ENVTYPE = "Statement"
GOSUB CHECK.STATUS
*
```

Before you can execute the SQL statement, you must bind all parameters. You normally bind parameters when data will be fetched several times. Using the statement environment, the program translates and places the following UniBasic variables into a field element:

- Parameter 1 – The name of the statement environment.
- Parameter 2 – The number of the table element the program references
- Parameter 3 – The type of conversion
- Parameter 4 – The SQL data type
- Parameter 5 – Scale or size of the field, if applicable
- Parameter 6 – Precision, if applicable
- Parameter 7 – UniBasic variable to load

```
STATUS = SQLBindParameter(statement.env, 1, SQL.B.BASIC, SQL.CHAR, 10, 0, I

*
```

```
Fn = "SQLBindParameter"
ENVTYPE = "Statement"
GOSUB CHECK.STATUS
*
STATUS = SQLBindParameter(statement.env, 2, SQL.B.BASIC, SQL.CHAR, 10, 0, N

*
GOSUB CHECK.STATUS

*
STATUS = SQLBindParameter(statement.env, 3, SQL.B.BASIC, SQL.CHAR, 10, 0, G

*
GOSUB CHECK.STATUS

*
STATUS = SQLBindParameter(statement.env, 4, SQL.B.BASIC, SQL.CHAR, 15, 0, C

*
GOSUB CHECK.STATUS
*
ROWNUM = 0
*
```

The program now starts the loop to retrieve data for the SQL database.

```
*
SELECT EMPFILE
LOOP
READNEXT ID ELSE ID = ""
WHILE ID NE "" DO
ROWNUM += 1
READ REC FROM EMPFILE, ID ELSE STOP "Error reading local EMPFILE file"
*
```

The program now assigns variables for the necessary fields.

```
*
NAME = REC<1>
GRADE = REC<2>
CITY = REC<3>
CRT "Loading row ":ROWNUM:" from EMPFILE"
*
```

The SQLExecute statement knows what to execute from previously preparing and binding the statement environment.

```
*
STATUS = SQLExecute(statement.env)
*
Fn = "SQLExecute"
ENVTYPE = "Statement"
GOSUB CHECK.STATUS
REPEAT
RETURN
*
CONFIRM.TABLE:
*
MODULE = "CONFIRM.TABLE"
form = "T###########"
dash = "----------"
```

The program now creates a UniBasic variable for executing an SQL statement on the new table.

```
*
SelectStmt = "SELECT ID, NAME, GRADE, CITY FROM EMPTABLE"
```

The program executes the SQL select statement through the statement environment.

```
*
STATUS = SQLExecDirect(statement.env,SelectStmt)
*
Fn = "SQLExecDirect"
ENVTYPE = "Statement"
GOSUB CHECK.STATUS
```

The following SQLBindCol statements bind the statement environment to return values from elements 1 through 4 (parameters 1 and 2). SQL.B.CHAR is used when translating SQL.CHAR data types. SQL.B.NUMBER is used when translating SQL.NUMERIC data types. The value returned is parameter 4.

```
STATUS = SQLBindCol(statement.env, 1, SQL.B.CHAR, ID.RET)
*
Fn = "SQLBindCol"
GOSUB CHECK.STATUS
*
STATUS = SQLBindCol(statement.env, 2, SQL.B.CHAR, NAME.RET)
GOSUB CHECK.STATUS
*
STATUS = SQLBindCol(statement.env, 3, SQL.B.NUMBER, GRADE.RET)
GOSUB CHECK.STATUS
STATUS = SQLBindCol(statement.env, 4, SQL.B.CHAR, CITY.RET)
GOSUB CHECK.STATUS
*
PRINT "ID" form:"NAME" form:"GRADE" form :"CITY" form
PRINT dash form:dash form:dash form :dash form
STATUS = 0
Fn = "SQLFetch"
ENVTYPE = "Statement"
```

As the program loops, it gets the next row of data from columns 1 - 4, populating the new variables.

```
LOOP
WHILE STATUS <> SQL.NO.DATA.FOUND DO
*
*
STATUS = SQLFetch(statement.env)
*
GOSUB CHECK.STATUS
IF STATUS <> SQL.NO.DATA.FOUND
THEN
CRT ID.RET form:NAME.RET form:GRADE.RET form:CITY.RET
END
REPEAT
*
ENVTYPE = "Statement"
GOSUB CHECK.STATUS
RETURN
```

Next the `SQLFreeStmt` uses the SQL.UNBIND option to release all bound statements for this statement environment.

```
FREE.ENV:
*
```

```
*
STATUS = SQLFreeStmt(statement.env, SQL.UNBIND)
*
```

The `SQLFreeStmt` with the SQL.DROP options releases the current statement environment.

```
*
STATUS = SQLFreeStmt(statement.env, SQL.DROP)
```

The `SQLDisconnect` statement disconnects the connection environment from the database.

```
*
STATUS = SQLDisconnect(connection.env)
```

Now the `SQLFreeConnect` statement releases the connection environment and its resources.

```
*
STATUS = SQLFreeConnect(connection.env)
```

The final step to release the UniData BCI environment and its resources is the execute the `SQLFreeEnv` statement.

```
*
STATUS = SQLFreeEnv(database.env)
*
RETURN
*
```

The following example illustrates an error handling routine.

```
CHECK.STATUS:
EXIT.PROGRAM = 0
BEGIN CASE
        CASE STATUS = SQL.ERROR
                RETMSG = "STATUS is ":STATUS:" -> Error Occurred."
                EXIT.PROGRAM = 1
        CASE STATUS = SQL.INVALID.HANDLE
                RETMSG = "STATUS is ":STATUS:" -> Invalid Connection Handle."
                EXIT.PROGRAM = 1
        CASE STATUS = SQL.NEED.DATA
                RETMSG = "STATUS is ":STATUS:" -> Data Required."
        CASE STATUS = SQL.NO.DATA.FOUND
                RETMSG = "STATUS is ":STATUS:" -> No Data Found."
        CASE STATUS = SQL.SUCCESS
                RETMSG = "STATUS is ":STATUS:" -> Successful."
        CASE STATUS = SQL.SUCCESS.WITH.INFO
                RETMSG = "STATUS is ":STATUS:" -> Successful With Info."
        CASE 1
                RETMSG = "STATUS is ":STATUS:" -> Unknown Status."
                EXIT.PROGRAM = 1
END CASE
*
BEGIN CASE
        CASE ENVTYPE = "Connection"
                CONNECT.VAR = connection.env
        CASE ENVTYPE = "Statement"
                CONNECT.VAR = statement.env
        CASE ENVTYPE = "Database"
                CONNECT.VAR = database.env
END CASE
*
```

```
IF EXIT.PROGRAM THEN
*
* ------------------------------------------------------------------------
* SQLError returns more information about errors.
* ========================================================================
*
ERROR.STATUS = SQLERROR(-1,CONNECT.VAR,-1,STATE,NATCODE,ERRTXT)
        CRT "In Module: ":MODULE
        CRT "Called ":Fn:" function in ":ENVTYPE:" environment."
        CRT RETMSG
        CRT "SQLSTATE , NATCOD are:" : STATE:" , ":NATCODE
        CRT "Error text is ":ERRTXT
RETURN TO END.OF.PROGRAM
        END
        RETURN
```

The following example illustrates the output from the BCI.DEMO program.

```
:RUN BP BCI.DEMO
Attempting connect to Server7 with user id sa
Deleting local EMPFILE file
Deleting file D_EMPFILE.
Deleting file EMPFILE.
Create file D_EMPFILE, modulo/1,blocksize/1024
Hash type = 0
Create file EMPFILE, modulo/3,blocksize/1024
Hash type = 0
Added "@ID", the default record for UniData to DICT EMPFILE.
EMPFILE is cleared.
Dropping EMPTABLE table in Server7
Creating EMPTABLE table in Server7
Loading row 1 from EMPFILE
Loading row 2 from EMPFILE
Loading row 3 from EMPFILE
Loading row 4 from EMPFILE
Loading row 5 from EMPFILE
ID      NAME    GRADE   CITY
---------------------------------------
E3      Carmen 13   Vienna
E1      Alice   12  Deale
E4      Don         12  Deale
E2      Betty   10  Vienna
E5      Ed      13  Akron
Exiting bcidemo
:
```

Appendix C: Error codes

The following table lists the `SQLSTATE` values and the corresponding messages they generate.

SQLSTATE	Message
00000	Successful completion
01002	Disconnect error
01004	Data truncated
07001	Wrong number of parameters
07006	Restricted data type attribute violation
08001	Unable to connect to data source
08002	Connection in use
08003	Connection not open
08007	Connection failure during transaction
08S01	Communication link failure
21S01	Insert value list does not match column list
21S02	Degree of derived table does not match column list
22001	String data right truncation
22003	Numeric value out of range
22005	Error in assignment
22008	Datetime field overflow
23000	Integrity constraint violation
24000	Invalid cursor state
25000	Invalid transaction state
34000	Invalid cursor name
3C000	Duplicate cursor name
40001	Serialization failure
42000	Syntax error or access violation
IM001	Driver does not support this function
IM002	Data source name not found and no default driver specified
IM003	Specified driver could not be loaded
IM980	Remote password is required
IM981	Multivalued data present, single result returned
IM982	Remote user ID is required
IM982	Only a single environment variable can be allocated
S0001	Base table or view already exists
S0002	Base table not found
S0021	Column already exists
S0022	Column not found
S1000	General error
S1001	Memory allocation failure
S1002	Invalid column number
S1003	Program type out of range

SQLSTATE	Message
S1004	SQL data type out of range
S1009	Invalid argument value
S1010	Function sequence error
S1012	Invalid transaction operation code specified
S1015	No cursor name available
S1090	Invalid string or buffer length
S1091	Descriptor type out of range
S1092	Option type out of range
S1093	Invalid parameter number
S1095	Function type out of range
S1096	Information type out of range
S1C00	Driver not capable
S1C00	Driver does not support this function

Appendix D: The ODBC.H file

This appendix lists the contents of the ODBC.H file. The ODBC.H file defines the values of column attributes.

```
************************************************************************
*# pg ODBC.H
************************************************************************
*
* Header file for ODBC BASIC programs
*
* Module %M% Version %I% Date %H%
*
* (c) Copyright 2005-2012 Rocket Software, Inc.- All Rights Reserved
* This is unpublished proprietary source code of Rocket Software.
* The copyright notice above does not evidence any actual or intented
* publication of such source code.
*
************************************************************************
*
* Maintenence log - insert most recent change descriptions at top
*
* Date.... GTAR# WHO Description.....................................
* 10/27/98 23888 CSM Add SQL.LIC.DEV.SUBKEY for licensing
* 10/14/98 23801 SAP Change copyrights.
* 06/19/97 20748 MJC BCI settings for SQLGetInfo and SQLSetConnectOption
* 11/18/96 19547 MJC BCI settings for SQLTransact
* 11/18/96 19547 MJC BCI settings for AUTOCOMMIT
* 11/06/96 19512 ENF Add BCI settings for PARAMOPTONS
* 09/04/96 19182 MJC Add SQL.COLUMN.DISPLAY.SIZE
* 08/08/96 18994 ENF Add EMPTY.NULL, TX.PRIVATE
* 07/29/96 18758 MJC Add SQL.COLUMN.PRINT.RESULT as 1004
* 07/25/96 18854 DTM Changes for ODBC middleware project
* 07/23/96 18854 DTM Changes for ODBC middleware project
* 05/24/96 18519 HSB Define parameter types for SQLBindParameter
* 07/31/95 16901 MGM Fix num.sql.types
* 07/25/95 16901 MGM Also fix 16191
* 05/03/95 15921 ENF Add some new ColAttributes option support
* 12/01/93 12544 ENF Added SQL.DATEFORM and SQL.DATEPREC
* 10/05/93 12380 ENF Initial submission
*
************************************************************************
* SQL Error RETCODES and defines.
EQU SQL.ERROR                           TO  -1
EQU SQL.INVALID.HANDLE              TO  -2
EQU SQL.NEED.DATA                       TO  99
EQU SQL.NO.DATA.FOUND               TO  100
EQU SQL.SUCCESS                         TO  0
EQU SQL.SUCCESS.WITH.INFO           TO  1
EQU SQL.NULL.HENV                       TO  -1
EQU SQL.NULL.HDBC                       TO  -1
EQU SQL.NULL.HSTMT                      TO  -1
EQU SQL.NULL.DATA                       TO  -1

* SQLColAttributes defines

EQU SQL.COLUMN.COUNT                    TO  1
EQU SQL.COLUMN.NAME                     TO  2
EQU SQL.COLUMN.TYPE                     TO  3
EQU SQL.COLUMN.LENGTH               TO  4
```

```
EQU SQL.COLUMN.PRECISION            TO  5
EQU SQL.COLUMN.SCALE                    TO  6
EQU SQL.COLUMN.DISPLAYSIZE      TO  7
EQU SQL.COLUMN.DISPLAY.SIZE         TO  7
EQU SQL.COLUMN.NULLABLE             TO  8
EQU SQL.COLUMN.UNSIGNED             TO  9
EQU SQL.COLUMN.MONEY                    TO  10
EQU SQL.COLUMN.UPDATABLE            TO  11
EQU SQL.COLUMN.AUTO.INCREMENT   TO  12
EQU SQL.COLUMN.CASE.SENSITIVE   TO  13
EQU SQL.COLUMN.SEARCHABLE           TO  14
EQU SQL.COLUMN.TYPE.NAME            TO  15
EQU SQL.COLUMN.TABLE.NAME           TO  16
EQU SQL.COLUMN.OWNER.NAME           TO  17
EQU SQL.COLUMN.QUALIFIER.NAME   TO  18
EQU SQL.COLUMN.LABEL                    TO  19
EQU SQL.COLUMN.MULTIVALUED      TO  1001
EQU SQL.COLUMN.FORMAT               TO  1002
EQU SQL.COLUMN.CONVERSION           TO 1003
EQU SQL.COLUMN.PRINT.RESULT         TO 1004

* SQLColAttributes subdefines for SQL.COLUMN.UPDATABLE

EQU SQL.ATTR.READONLY               TO 0
EQU SQL.ATTR.WRITE                      TO 1
EQU SQL.ATTR.READWRITE.UNKNOWN TO 2

* SQLColAttributes subdefines for SQL.COLUMN.SEARCHABLE

EQU SQL.UNSEARCHABLE                    TO 0
EQU SQL.LIKE.ONLY                       TO 1
EQU SQL.ALL.EXCEPT.LIKE             TO 2
EQU SQL.SEARCHABLE                      TO 3

* SQLSetConnectOption defines

EQU SQL.AUTOCOMMIT                      TO 102
EQU SQL.USE.ODBC.PRECISION      TO 999
EQU SQL.TRUNC.ROUND                     TO 998
EQU SQL.SEND.TRUNC.ROUND            TO 997
EQU SQL.OS.UID                              TO 996
EQU SQL.OS.PWD                              TO 995
EQU SQL.DATEFORM                            TO 994
EQU SQL.DATEPREC                            TO 993
EQU SQL.AUTOCOMMIT.OFF              TO 0
EQU SQL.AUTOCOMMIT.ON               TO 1
EQU SQL.EMPTY.NULL                      TO 1003
EQU SQL.EMPTY.NULL.ON               TO 1
EQU SQL.EMPTY.NULL.OFF              TO 0
EQU SQL.TX.PRIVATE                      TO 1004
EQU SQL.TX.PRIVATE.ON               TO 1
EQU SQL.TX.PRIVATE.OFF              TO 0
EQU SQL.UVNLS.MAP                       TO 1005
EQU SQL.UVNLS.LOCALE                    TO 1006
EQU SQL.UVNLS.LC.TIME               TO 1007
EQU SQL.UVNLS.LC.NUMERIC            TO 1008
EQU SQL.UVNLS.LC.MONETARY           TO 1009
EQU SQL.UVNLS.LC.CTYPE              TO 1010
EQU SQL.UVNLS.LC.COLLATE            TO 1011
EQU SQL.UVNLS.LC.ALL                    TO 1012
EQU SQL.UVNLS.SQL.NULL              TO 1013
EQU SQL.UVNLS.TEXT.MARK             TO 1014
```

```
EQU SQL.UVNLS.SUBVALUE.MARK         TO 1015
EQU SQL.UVNLS.VALUE.MARK            TO 1016
EQU SQL.UVNLS.FIELD.MARK            TO 1017
EQU SQL.UVNLS.ITEM.MARK             TO 1018
EQU SQL.LIC.DEV.SUBKEY              TO 1019

* SQLFreeStmt option defines

EQU SQL.CLOSE                              TO 1
EQU SQL.DROP                               TO 2
EQU SQL.UNBIND                             TO 3
EQU SQL.RESET.PARAMS                    TO 4

* Define all SQL data types
* and those that we support

EQU SQL.CHAR                               TO 1
EQU SQL.NUMERIC                            TO 2
EQU SQL.DECIMAL                            TO 3
EQU SQL.INTEGER                            TO 4
EQU SQL.SMALLINT                           TO 5
EQU SQL.FLOAT                              TO 6
EQU SQL.REAL                               TO 7
EQU SQL.DOUBLE                             TO 8
EQU SQL.DATE                               TO 9
EQU SQL.TIME                               TO 10
EQU SQL.TIMESTAMP                       TO 11
EQU SQL.VARCHAR                            TO 12
EQU SQL.LONGVARCHAR                     TO -1
EQU SQL.BINARY                             TO -2
EQU SQL.VARBINARY                       TO -3
EQU SQL.LONGVARBINARY               TO -4
EQU SQL.BIGINT                             TO -5
EQU SQL.TINYINT                            TO -6
EQU SQL.BIT                                    TO -7
EQU NUM.SQL.TYPES                       TO 19

* Define ODBC conception of display size
* for the various data types

EQU SQL.CHAR.DSPSIZE                    TO 0
EQU SQL.VARCHAR.DSPSIZE             TO 0
EQU SQL.DECIMAL.DSPSIZE             TO 2
EQU SQL.NUMERIC.DSPSIZE             TO 2
EQU SQL.SMALLINT.DSPSIZE            TO 6
EQU SQL.INTEGER.DSPSIZE             TO 11
EQU SQL.REAL.DSPSIZE                    TO 13
EQU SQL.FLOAT.DSPSIZE               TO 22
EQU SQL.DOUBLE.DSPSIZE              TO 22
EQU SQL.DATE.DSPSIZE                    TO 10
EQU SQL.TIME.DSPSIZE                    TO 8

* Define ODBC conception of precision
* for the various data types

EQU SQL.CHAR.PRECISION              TO 254
EQU SQL.VARCHAR.PRECISION           TO 254
EQU SQL.DECIMAL.PRECISION           TO 15
EQU SQL.NUMERIC.PRECISION           TO 15
EQU SQL.SMALLINT.PRECISION      TO 5
EQU SQL.INTEGER.PRECISION           TO 10
EQU SQL.REAL.PRECISION              TO 7
```

```
EQU SQL.FLOAT.PRECISION             TO 15
EQU SQL.DOUBLE.PRECISION            TO 15
EQU SQL.DATE.PRECISION              TO 10
EQU SQL.TIME.PRECISION              TO 8

* Valid BASIC data types

EQU SQL.B.BASIC                             TO 100
EQU SQL.B.INTDATE                       TO 101
EQU SQL.B.NUMBER                            TO 102
EQU SQL.B.INTTIME                       TO 103
EQU SQL.B.CHAR                              TO 1
EQU SQL.B.DEFAULT                       TO 99

* Define return valued for
* Describe and ColAttributes

EQU SQL.NO.NULLS                            TO 0
EQU SQL.NULLABLE                            TO 1
EQU SQL.NULLABLE.UNKNOWN            TO 2

* Define parameter types for SQLBindParameter (SQLSetParam)

EQU SQL.PARAM.INPUT                     TO 1
EQU SQL.PARAM.INPUT.OUTPUT      TO 2
EQU SQL.PARAM.OUTPUT                    TO 4

* DTM Added for BCI/Datastage - SQLGetInfo

EQU SQL.ACTIVE.CONNECTIONS      TO 0
EQU SQL.ACTIVE.STATEMENTS           TO 1
EQU SQL.DATA.SOURCE.NAME            TO 2
EQU SQL.DRIVER.HDBC                     TO 3
EQU SQL.DRIVER.HENV                     TO 4
EQU SQL.DRIVER.HSTMT                    TO 5
EQU SQL.DRIVER.NAME                     TO 6
EQU SQL.DRIVER.VER                      TO 7
EQU SQL.FETCH.DIRECTION             TO 8
EQU SQL.ODBC.API.CONFORMANCE    TO 9
EQU SQL.ODBC.VER                            TO 10
EQU SQL.ROW.UPDATES                         TO 11
EQU SQL.ODBC.SAG.CLI.CONFORMANCE TO 12
EQU SQL.SERVER.NAME                         TO 13
EQU SQL.SEARCH.PATTERN.ESCAPE   TO 14
EQU SQL.ODBC.SQL.CONFORMANCE    TO 15
EQU SQL.DATABASE.NAME               TO 16
EQU SQL.DBMS.NAME                       TO 17
EQU SQL.DBMS.VER                            TO 18
EQU SQL.ACCESSIBLE.TABLES           TO 19
EQU SQL.ACCESSIBLE.PROCEDURES   TO 20
EQU SQL.PROCEDURES                      TO 21
EQU SQL.CONCAT.NULL.BEHAVIOR    TO 22
EQU SQL.CURSOR.COMMIT.BEHAVIOR TO 23
EQU SQL.CURSOR.ROLLBACK.BEHAVIOR TO 24
EQU SQL.DATA.SOURCE.READ.ONLY   TO 25
EQU SQL.DEFAULT.TXN.ISOLATION   TO 26
EQU SQL.EXPRESSIONS.IN.ORDERBY TO 27
EQU SQL.IDENTIFIER.CASE             TO 28
EQU SQL.IDENTIFIER.QUOTE.CHAR   TO 29
EQU SQL.MAX.COLUMN.NAME.LEN         TO 30
EQU SQL.MAX.CURSOR.NAME.LEN         TO 31
EQU SQL.MAX.OWNER.NAME.LEN      TO 32
```

```
EQU SQL.MAX.PROCEDURE.NAME.LEN TO 33
EQU SQL.MAX.QUALIFIER.NAME.LEN TO 34
EQU SQL.MAX.TABLE.NAME.LEN      TO 35
EQU SQL.MULT.RESULT.SETS            TO 36
EQU SQL.MULTIPLE.ACTIVE.TXN         TO 37
EQU SQL.OUTER.JOINS                     TO 38
EQU SQL.OWNER.TERM                      TO 39
EQU SQL.PROCEDURE.TERM              TO 40
EQU SQL.QUALIFIER.NAME.SEPARATOR TO 41
EQU SQL.QUALIFIER.TERM              TO 42
EQU SQL.SCROLL.CONCURRENCY      TO 43
EQU SQL.SCROLL.OPTIONS              TO 44
EQU SQL.TABLE.TERM                      TO 45
EQU SQL.TXN.CAPABLE                     TO 46
EQU SQL.USER.NAME                       TO 47
EQU SQL.CONVERT.FUNCTIONS           TO 48
EQU SQL.NUMERIC.FUNCTIONS           TO 49
EQU SQL.STRING.FUNCTIONS            TO 50
EQU SQL.SYSTEM.FUNCTIONS            TO 51
EQU SQL.TIMEDATE.FUNCTIONS      TO 52
EQU SQL.CONVERT.BIGINT              TO 53
EQU SQL.CONVERT.BINARY              TO 54
EQU SQL.CONVERT.BIT                     TO 55
EQU SQL.CONVERT.CHAR                    TO 56
EQU SQL.CONVERT.DATE                    TO 57
EQU SQL.CONVERT.DECIMAL             TO 58
EQU SQL.CONVERT.DOUBLE              TO 59
EQU SQL.CONVERT.FLOAT               TO 60
EQU SQL.CONVERT.INTEGER             TO 61
EQU SQL.CONVERT.LONGVARCHAR         TO 62
EQU SQL.CONVERT.NUMERIC             TO 63
EQU SQL.CONVERT.REAL                    TO 64
EQU SQL.CONVERT.SMALLINT            TO 65
EQU SQL.CONVERT.TIME                    TO 66
EQU SQL.CONVERT.TIMESTAMP           TO 67
EQU SQL.CONVERT.TINYINT             TO 68
EQU SQL.CONVERT.VARBINARY           TO 69
EQU SQL.CONVERT.VARCHAR             TO 70
EQU SQL.CONVERT.LONGVARBINARY   TO 71
EQU SQL.TXN.ISOLATION.OPTION    TO 72
EQU SQL.ODBC.SQL.OPT.IEF            TO 73
EQU SQL.CORRELATION.NAME            TO 74
EQU SQL.NON.NULLABLE.COLUMNS    TO 75
EQU SQL.DRIVER.HLIB                     TO 76
EQU SQL.DRIVER.ODBC.VER             TO 77
EQU SQL.LOCK.TYPES                      TO 78
EQU SQL.POS.OPERATIONS              TO 79
EQU SQL.POSITIONED.STATEMENTS   TO 80
EQU SQL.GETDATA.EXTENSIONS      TO 81
EQU SQL.BOOKMARK.PERSISTENCE    TO 82
EQU SQL.STATIC.SENSITIVITY      TO 83
EQU SQL.FILE.USAGE                      TO 84
EQU SQL.NULL.COLLATION              TO 85
EQU SQL.ALTER.TABLE                     TO 86
EQU SQL.COLUMN.ALIAS                    TO 87
EQU SQL.GROUP.BY                            TO 88
EQU SQL.KEYWORDS                            TO 89
EQU SQL.ORDER.BY.COLUMNS.IN.SELECT TO 90
EQU SQL.OWNER.USAGE                     TO 91
EQU SQL.QUALIFIER.USAGE             TO 92
EQU SQL.QUOTED.IDENTIFIER.CASE TO 93
EQU SQL.SPECIAL.CHARACTERS      TO 94
```

```
EQU SQL.SUBQUERIES                      TO 95
EQU SQL.UNION                               TO 96
EQU SQL.MAX.COLUMNS.IN.GROUP.BY     TO 97
EQU SQL.MAX.COLUMNS.IN.INDEX        TO 98
EQU SQL.MAX.COLUMNS.IN.ORDER.BY     TO 99
EQU SQL.MAX.COLUMNS.IN.SELECT       TO 100
EQU SQL.MAX.COLUMNS.IN.TABLE        TO 101
EQU SQL.MAX.INDEX.SIZE                  TO 102
EQU SQL.MAX.ROW.SIZE.INCLUDES.LONG TO 103
EQU SQL.MAX.ROW.SIZE                        TO 104
EQU SQL.MAX.STATEMENT.LEN               TO 105
EQU SQL.MAX.TABLES.IN.SELECT        TO 106
EQU SQL.MAX.USER.NAME.LEN               TO 107
EQU SQL.MAX.CHAR.LITERAL.LEN        TO 108
EQU SQL.TIMEDATE.ADD.INTERVALS  TO 109
EQU SQL.TIMEDATE.DIFF.INTERVALS     TO 110
EQU SQL.NEED.LONG.DATA.LEN          TO 111
EQU SQL.MAX.BINARY.LITERAL.LEN  TO 112
EQU SQL.LIKE.ESCAPE.CLAUSE          TO 113
EQU SQL.QUALIFIER.LOCATION          TO 114

* SQL_ALTER_TABLE bitmasks *

EQU SQL.AT.ADD.COLUMN               TO 1
EQU SQL.AT.DROP.COLUMN              TO 2

* SQL_BOOKMARK_PERSISTENCE bitmasks *

EQU SQL.BP.CLOSE                            TO 1
EQU SQL.BP.DELETE                           TO 2
EQU SQL.BP.DROP                             TO 4
EQU SQL.BP.TRANSACTION              TO 8
EQU SQL.BP.UPDATE                       TO 16
EQU SQL.BP.OTHER.HSTMT              TO 32
EQU SQL.BP.SCROLL                       TO 64

* SQL_CONCAT_NULL_BEHAVIOR values *

EQU SQL.CB.NULL                             TO 0
EQU SQL.CB.NON.NULL                     TO 1

* SQL_CURSOR_COMMIT_BEHAVIOR values *
* SQL_CURSOR_ROLLBACK_BEHAVIOR values *

EQU SQL.CB.DELETE                       TO 0
EQU SQL.CB.CLOSE                            TO 1
EQU SQL.CB.PRESERVE                     TO 2

* SQL_CORRELATION_NAME values *

EQU SQL.CN.NONE                             TO 0
EQU SQL.CN.DIFFERENT                    TO 1
EQU SQL.CN.ANY                              TO 2

* SQL_CONVERT_<.> bitmasks *

EQU SQL.CVT.CHAR                            TO 1
EQU SQL.CVT.NUMERIC                     TO 2
EQU SQL.CVT.DECIMAL                     TO 4
EQU SQL.CVT.INTEGER                     TO 8
EQU SQL.CVT.SMALLINT                    TO 16
EQU SQL.CVT.FLOAT                       TO 32
```

```
EQU SQL.CVT.REAL                            TO 64
EQU SQL.CVT.DOUBLE                      TO 128
EQU SQL.CVT.VARCHAR                     TO 256
EQU SQL.CVT.LONGVARCHAR             TO 512
EQU SQL.CVT.BINARY                      TO 1024
EQU SQL.CVT.VARBINARY               TO 2048
EQU SQL.CVT.BIT                             TO 4096
EQU SQL.CVT.TINYINT                     TO 8192
EQU SQL.CVT.BIGINT                      TO 16384
EQU SQL.CVT.DATE                            TO 32768
EQU SQL.CVT.TIME                            TO 65536
EQU SQL.CVT.TIMESTAMP               TO 131072
EQU SQL.CVT.LONGVARBINARY           TO 262144

* SQL_FETCH_DIRECTION bitmask *

EQU SQL.FD.FETCH.NEXT               TO 1
EQU SQL.FD.FETCH.FIRST              TO 2
EQU SQL.FD.FETCH.LAST               TO 4
EQU SQL.FD.FETCH.PRIOR              TO 8
EQU SQL.FD.FETCH.ABSOLUTE           TO 16
EQU SQL.FD.FETCH.RELATIVE           TO 32
EQU SQL.FD.FETCH.RESUME             TO 64
EQU SQL.FD.FETCH.BOOKMARK           TO 128

* SQL_FILE_USAGE values *

EQU SQL.FILE.NOT.SUPPORTED      TO 0
EQU SQL.FILE.TABLE                      TO 1
EQU SQL.FILE.QUALIFIER              TO 2

* SQL_CONVERT_FUNCTIONS bitmask *

EQU SQL.FN.CVT.CONVERT              TO 1

* SQL_NUMERIC_FUNCTIONS bitmask *

EQU SQL.FN.NUM.ABS                      TO 1
EQU SQL.FN.NUM.ACOS                     TO 2
EQU SQL.FN.NUM.ASIN                     TO 4
EQU SQL.FN.NUM.ATAN                     TO 8
EQU SQL.FN.NUM.ATAN2                    TO 16
EQU SQL.FN.NUM.CEILING              TO 32
EQU SQL.FN.NUM.COS                      TO 64
EQU SQL.FN.NUM.COT                      TO 128
EQU SQL.FN.NUM.EXP                      TO 256
EQU SQL.FN.NUM.FLOOR                    TO 512
EQU SQL.FN.NUM.LOG                      TO 1024
EQU SQL.FN.NUM.MOD                      TO 2048
EQU SQL.FN.NUM.SIGN                     TO 4096
EQU SQL.FN.NUM.SIN                      TO 8192
EQU SQL.FN.NUM.SQRT                     TO 16384
EQU SQL.FN.NUM.TAN                      TO 32768
EQU SQL.FN.NUM.PI                       TO 65536
EQU SQL.FN.NUM.RAND                     TO 131072
EQU SQL.FN.NUM.DEGREES              TO 262144
EQU SQL.FN.NUM.LOG10                    TO 524288
EQU SQL.FN.NUM.POWER                    TO 1048576
EQU SQL.FN.NUM.RADIANS              TO 2097152
EQU SQL.FN.NUM.ROUND                    TO 4194304
EQU SQL.FN.NUM.TRUNCATE             TO 8388608
```

```
* SQL_STRING_FUNCTIONS bitmask *

EQU SQL.FN.STR.CONCAT               TO 1
EQU SQL.FN.STR.INSERT               TO 2
EQU SQL.FN.STR.LEFT                    TO 4
EQU SQL.FN.STR.LTRIM                   TO 8
EQU SQL.FN.STR.LENGTH               TO 16
EQU SQL.FN.STR.LOCATE               TO 32
EQU SQL.FN.STR.LCASE                   TO 64
EQU SQL.FN.STR.REPEAT               TO 128
EQU SQL.FN.STR.REPLACE              TO 256
EQU SQL.FN.STR.RIGHT                   TO 512
EQU SQL.FN.STR.RTRIM                   TO 1024
EQU SQL.FN.STR.SUBSTRING            TO 2048
EQU SQL.FN.STR.UCASE                   TO 4096
EQU SQL.FN.STR.ASCII                   TO 8192
EQU SQL.FN.STR.CHAR                    TO 16384
EQU SQL.FN.STR.DIFFERENCE           TO 32768
EQU SQL.FN.STR.LOCATE.2             TO 65536
EQU SQL.FN.STR.SOUNDEX              TO 131072
EQU SQL.FN.STR.SPACE                   TO 262144

* SQL_SYSTEM_FUNCTIONS bitmask *

EQU SQL.FN.SYS.USERNAME             TO 1
EQU SQL.FN.SYS.DBNAME               TO 2
EQU SQL.FN.SYS.IFNULL               TO 4

* SQL_TIMEDATE bitmask *

EQU SQL.FN.TD.NOW                      TO 1
EQU SQL.FN.TD.CURDATE               TO 2
EQU SQL.FN.TD.DAYOFMONTH            TO 4
EQU SQL.FN.TD.DAYOFWEEK             TO 8
EQU SQL.FN.TD.DAYOFYEAR             TO 16
EQU SQL.FN.TD.MONTH                    TO 32
EQU SQL.FN.TD.QUARTER               TO 64
EQU SQL.FN.TD.WEEK                     TO 128
EQU SQL.FN.TD.YEAR                     TO 256
EQU SQL.FN.TD.CURTIME               TO 512
EQU SQL.FN.TD.HOUR                     TO 1024
EQU SQL.FN.TD.MINUTE                   TO 2048
EQU SQL.FN.TD.SECOND                   TO 4096
EQU SQL.FN.TD.TIMESTAMPADD      TO 8192
EQU SQL.FN.TD.TIMESTAMPDIFF         TO 16384
EQU SQL.FN.TD.DAYNAME               TO 32768
EQU SQL.FN.TD.MONTHNAME             TO 65536

* SQL_TIMEDATE_ADD_INTERVALS bitmask *
* SQL_TIMEDATE_DIFF_INTERVALS bitmask *

EQU SQL.FN.TSI.FRAC.SECOND      TO 1
EQU SQL.FN.TSI.SECOND               TO 2
EQU SQL.FN.TSI.MINUTE               TO 4
EQU SQL.FN.TSI.HOUR                    TO 8
EQU SQL.FN.TSI.DAY                     TO 16
EQU SQL.FN.TSI.WEEK                    TO 32
EQU SQL.FN.TSI.MONTH                   TO 64
EQU SQL.FN.TSI.QUARTER              TO 128
EQU SQL.FN.TSI.YEAR                    TO 256

* SQL_GROUP_BY values *
```

```
EQU SQL.GB.NOT.SUPPORTED                    TO 0
EQU SQL.GB.GROUP.BY.EQUALS.SELECT   TO 1
EQU SQL.GB.GROUP.BY.CONTAINS.SELECT TO 2
EQU SQL.GB.NO.RELATION                      TO 3

* SQL_GETDATA_EXTENSIONS values *

EQU SQL.GD.ANY.COLUMN               TO 1
EQU SQL.GD.ANY.ORDER                    TO 2
EQU SQL.GD.BLOCK                            TO 4
EQU SQL.GD.BOUND                            TO 8

* SQL_IDENTIFIER_CASE values *
* SQL_QUOTED_IDENTIFIER values *

EQU SQL.IC.UPPER                            TO 1
EQU SQL.IC.LOWER                            TO 2
EQU SQL.IC.SENSITIVE                    TO 3
EQU SQL.IC.MIXED                            TO 4

* SQL_LOCK_TYPES bitmask *

EQU SQL.LCK.NO.CHANGE               TO 1
EQU SQL.LCK.EXCLUSIVE               TO 2
EQU SQL.LCK.UNLOCK                      TO 4

* SQL_NULL_COLLATION values *

EQU SQL.NC.HIGH                                 TO 0
EQU SQL.NC.LOW                                  TO 1
EQU SQL.NC.START                                TO 2
EQU SQL.NC.END                                  TO 4

* SQL_NON_NULLABLE_COLUMNS values *

EQU SQL.NNC.NULL                                TO 0
EQU SQL.NNC.NON.NULL                        TO 1

* SQL_ODBC_API_CONFORMANCE *

EQU SQL.OAC.NONE                                TO 0
EQU SQL.OAC.LEVEL1                          TO 1
EQU SQL.OAC.LEVEL2                          TO 2

* SQL_ODBC_SQL_CONFORMANCE values *

EQU SQL.OSC.MINIMUM                         TO 0
EQU SQL.OSC.CORE                                TO 1
EQU SQL.OSC.EXTENDED                        TO 2

* SQL_ODBC_SAG_CLI_CONFORMANCE values *

EQU SQL.OSCC.NOT.COMPLIANT          TO 0
EQU SQL.OSCC.COMPLIANT                  TO 1

* SQL_OWNER_USAGE bitmask *

EQU SQL.OU.DML.STATEMENTS               TO 1
EQU SQL.OU.PROCEDURE.INVOCATION     TO 2
EQU SQL.OU.TABLE.DEFINITION             TO 4
EQU SQL.OU.INDEX.DEFINITION             TO 8
```

```
EQU SQL.OU.PRIVILEGE.DEFINITION TO 16

* SQL_POS_OPERATIONS *

EQU SQL.POS.POSITION                        TO 1
EQU SQL.POS.REFRESH                         TO 2
EQU SQL.POS.UPDATE                          TO 4
EQU SQL.POS.DELETE                          TO 8
EQU SQL.POS.ADD                                 TO 16

* SQL_POSITIONED_STATEMENTS bitmask *

EQU SQL.PS.POSITIONED.DELETE        TO 1
EQU SQL.PS.POSITIONED.UPDATE        TO 2
EQU SQL.PS.SELECT.FOR.UPDATE        TO 4

* SQL_DEFAULT_TXN_ISOLATION bitmask *
* SQL_TXN_ISOLATION_OPTION bitmask *

EQU SQL.TXN.READ.UNCOMMITTED        TO 1
EQU SQL.TXN.READ.COMMITTED          TO 2
EQU SQL.TXN.REPEATABLE.READ             TO 4
EQU SQL.TXN.SERIALIZABLE                TO 8
EQU SQL.TXN.VERSIONING                  TO 16
EQU SQL.TXN.CURRENT                         TO 42

* SQL_QUALIFIER_LOCATION values *

EQU SQL.QL.START                                TO 1
EQU SQL.QL.END                                  TO 2

* SQL_QUALIFIER_USAGE bitmask *

EQU SQL.QU.DML.STATEMENTS               TO 1
EQU SQL.QU.PROCEDURE.INVOCATION     TO 2
EQU SQL.QU.TABLE.DEFINITION             TO 4
EQU SQL.QU.INDEX.DEFINITION             TO 8
EQU SQL.QU.PRIVILEGE.DEFINITION TO 16

* SQL_SCROLL_CONCURRENCY bitmask *

EQU SQL.SCCO.READ.ONLY                  TO 1
EQU SQL.SCCO.LOCK                           TO 2
EQU SQL.SCCO.OPT.ROWVER                 TO 4
EQU SQL.SCCO.OPT.VALUES                 TO 8

* SQL_SCROLL_OPTIONS bitmask *

EQU SQL.SO.FORWARD.ONLY                 TO 1
EQU SQL.SO.KEYSET.DRIVEN                TO 2
EQU SQL.SO.DYNAMIC                          TO 4
EQU SQL.SO.MIXED                                TO 8
EQU SQL.SO.STATIC                           TO 16

* SQL_STATIC_SENSITIVITY bitmask *

EQU SQL.SS.ADDITIONS                        TO 1
EQU SQL.SS.DELETIONS                        TO 2
EQU SQL.SS.UPDATES                          TO 4

* SQL_SUBQUERIES bitmask *
```

```
EQU SQL.SQ.COMPARISON                   TO 1
EQU SQL.SQ.EXISTS                           TO 2
EQU SQL.SQ.IN                                   TO 4
EQU SQL.SQ.QUANTIFIED                   TO 8
EQU SQL.SQ.CORRELATED.SUBQUERIES TO 16

* SQL_TXN_CAPABLE values *

:EQU SQL.TC.NONE                        TO 0
EQU SQL.TC.DML                          TO 1
EQU SQL.TC.ALL                          TO 2
EQU SQL.TC.DDL.COMMIT           TO 3
EQU SQL.TC.DDL.IGNORE           TO 4

* SQL_UNION values *

EQU SQL.U.UNION                         TO 1
EQU SQL.U.UNION.ALL                 TO 2

* Additions for SQLSpecialColumns

EQU SQL.BEST.ROWID                  TO 1
EQU SQL.ROWVER                          TO 2
EQU SQL.SCOPE.CURROW                TO 0
EQU SQL.SCOPE.TRANSACTION       TO 1
EQU SQL.SCOPE.SESSION           TO 2
EQU SQL.PC.UNKNOWN                  TO 0
EQU SQL.PC.PSEUDO                   TO 1
EQU SQL.PC.NOT.PSEUDO           TO 2

* Additions for SQLStatistics

EQU SQL.INDEX.UNIQUE                TO 0
EQU SQL.INDEX.ALL                   TO 1
EQU SQL.QUICK                           TO 0
EQU SQL.ENSURE                          TO 1
EQU SQL.TABLE.STAT                  TO 0
EQU SQL.INDEX.CLUSTERED         TO 1
EQU SQL.INDEX.HASHED                TO 2
EQU SQL.INDEX.OTHER                 TO 3

* Additions for SQLParamOptions

EQU SQL.PARAMOPTIONS.SET        TO 0
EQU SQL.PARAMOPTIONS.READ       TO 1

* Additions for SQLTransact

EQU SQL.COMMIT                          TO 1
EQU SQL.ROLLBACK                        TO 2
#
```