HOW-TO
Access Data using BCI

Brian Leach Consulting Limited

http://www.brianleach.co.uk
Introduction

**The Basic Calling Interface (BCI)** provides a means of accessing external data sources from within your UniVerse or UniData applications using the ODBC standard. The BCI allows you to both read and write data to and from a wide variety of data sources, including many mainstream database types (Oracle, SQL Server, MySQL) and other ODBC compliant applications (an Excel spreadsheet, for example).

BCI offers a powerful alternative to using delimited files or other media to pass data to and from your MultiValue applications. BCI operates in real time, issuing standard SQL commands, so that changes can be viewed immediately. BCI speaks directly to the ODBC drivers, so that there is no need for additional processing to create or consume data in their native formats. And if you need to share or import data to and from your U2 system, BCI running from the server means that there is only one central point to administer the connections and any credentials required.

Despite this, BCI is rarely used on U2 applications. So why should that be?

There are several answers to that one:
- it is often overlooked or misunderstood
- it is perceived to be difficult
- it requires additional (costed) supporting software under UNIX
- the syntax is alien and looks frightening.

BCI is available for UniVerse and UniData platforms and in terms of usage it is virtually identical on both. In this article you will learn how to query a Microsoft Access database from within a UniVerse BASIC program. The article assumes that the database is running under Windows: for UNIX platforms there are some further steps required to get ODBC (which is not native to UNIX) up and running. For this article we will assume a U2 database running on a Windows server.

**NOTE**

You can use the Personal Edition of UniVerse or UniData to run these examples. Please see my website ([www.brianleach.co.uk](http://www.brianleach.co.uk)) for details on downloading these, as well as the samples and resources needed for this guide.

For this example we will be using the **Wychwood Audio Books** demonstration database available from [www.brianleach.co.uk](http://www.brianleach.co.uk). This is a database representing an audio bookstore, available in a number of flavours including UniVerse, UniData and here as a Microsoft Access database. Allow 1-2 hours to complete the workshops.
Notes

This page has been left blank for your notes.
Defining the Data Source

The first stage in using BCI is to define your data source to ODBC using the ODBC Manager in the Windows Control Panel. The ODBC sources can be defined in three locations, which provide different levels of visibility: for the current user (User DSN), for all users on the system (System DSN) or as a file that can be shared (File DSN). For sources available to BCI you should use the System DSN.

The ODBC Driver Manager is located in the Administration Tools section of the Windows control panel under the name Data Sources (ODBC).

If you are running a 64 bit version of Windows, there are two copies of the ODBC Manager: one for 64 bit displayed in the control panel, and one for 32 bit. If you are running a 32 bit version of UniVerse or UniData on a 64 bit version of Windows you will need to use the 32 bit version of the manager.

To run the 32 bit version on a 64 bit system, open a cmd shell and type the command:

```
c:\windows\syswow64\odbcad32.exe
```

For any other combination use the version in the control panel.

Open it up and select the System DSN Tab.

Each ODBC data source is defined using a unique name that identifies an entry holding the data source type, driver information and any additional details required by that driver. For this demonstration select Add to add a new data source and select the MS Access Database driver from the list.
Give the data source a name, for example, WYCHBOOKS. You will need to tell the driver where to locate the database you have downloaded, so click the Select button and navigate to the wychbooks.mdb database.

Save the definition.
Check the Definition

It is a good idea to test the definition before you go any further. A good tool for this is Microsoft Excel.

Open a new workbook and from the Data tab of the ribbon select From Other Source -> From Microsoft Query. The WYCHBOOKS data source should now appear in the list of databases: select this to open a list of the tables in the database.

You should find the main BOOK_TITLES table listed. Add the table to Columns in your Query:

Click through the options until the last page, where you can Add Data to Excel.
Setting up the BCI connection

You are now ready to access this through BCI. The next step differs slightly between UniVerse and UniData. On UniVerse, you must define the connection as a UniVerse data source using the uvodbc.config file. This is a text file held in the UniVerse home directory. (usually c:\u2\uv\uvodbc.config).

Simply add the lines:

<WYCHBOOKS>
DBMSTYPE = ODBC

The name of the data source (enclosed in angle braces) MUST match the name of the ODBC data source given above. Also note that there must be spaces around the equals (=) sign.

On UniData, which supports SQL connections only to SQL data sources and not to other UniVerse instances, this step is not required.

Starting with BCI

BCI is implemented as a series of BASIC functions and equates defined in UNIVERSE.INCLUDE ODBC.H (UniVerse) and INCLUDE ODBC.H (UniData). These all follow the same convention: they require and may set a number of arguments and they each return a status code that can be used to track if an error has occurred. For example, the SQLAllocEnv function sets a variable and returns a status code:

```
StatusCode = SQLAllocEnv(AVariable)
```

On UniData all these functions should be typed in upper case or compiled with the –I option:
```
StatusCode = SQLALLOCENV(aVariable).
```

This article follows the UniVerse format for legibility.

What makes the syntax unusual for U2 developers is that this is a very much 'behind the scenes' protocol. The database actually manages the SQL statements: allocating resources, setting parameters, fetching rows and assigning columns to variables. The same is true of many of the external protocols including socket, XML DOM and UDOs.

All of this is handled indirectly, the API returning handles or pointers to the structures created. If this seems unfamiliar, remember that it is not that dissimilar to working with file or select list variables.
Terminology:
A *handle* is a variable holding a pointer to a structure or object. These are values that are used to track objects that have been created internally by the BCI library. The values they hold are not meaningful if accessed directly and should not be altered by your code.

**Making a Connection**

The first step in any BCI routine is to initialize the BCI environment. This uses the `SQLAllocEnv()` function. This creates an environment for the BCI processing, and returns a variable that will be used to identify this environment in future connection calls:

\[
\text{StatusCode} = \text{SQLAllocEnv(hEnvironment)}
\]

Once the environment has been created it can host one or more connections to various data sources. Each connection is similarly allocated, and sets a variable that acts as an identifier. There are, in fact, three steps to making a data source connection: the connection is allocated, any options are set and finally the connection is made.

```
$INCLUDE UNIVERSE.INCLUDE ODBC.H

EQU DSN.NAME To "WYCHBOOKS"
EQU USER.NAME To ""
EQU PASSWORD To ""

* First initialize an ODBC environment
  Ok = SQLAllocEnv(hEnv)

* Allocate a connection environment
  Ok = SQLAllocConnect(hEnv, hConnect)

* And make the connection
  Ok = SQLConnect(hConnect, DSN.NAME, USER.NAME, PASSWORD)

Crt "Connected"
```

As with all good programming practice, resources that have been allocated need to be released when you have finished with them. The following three lines will clean up at the end of the program.

```
* Close all connections
  Ok = SQLDisconnect(hConnect)
  Ok = SQLFreeConnect(hConnect)
  Ok = SQLFreeEnv(hEnv)

(see bcidemo1)
```
Executing SQL Statements

There are two main methods by which you can execute an SQL statement against the database.

The simpler method is the `SQLExecDirect()` function, which passes a fully formed SQL statement to the ODBC driver. This is intended for two types of statements:

- Statements that do not return a set of data, such as a CREATE or GRANT statement.
- Statements that are executed only once.

Before you can execute an SQL statement you must allocate a statement environment. This creates a handle that is used to track the statement, for results or error processing. The handle is allocated using the `SQLAllocStmt()` function, which attaches the statement to your connection:

```c
Ok = SQLAllocStmt(hConnect, hStat)
```

As with all resources allocated, the statement handle should be released once you have finished with it. The `SQLFreeStmt()` function takes care of that:

```c
Ok = SQLFreeStmt(hStat, SQL.DROP)
```

The following example runs a select statement to return all of the authors in the database from the `BOOK_AUTHORS` table. The statement is executed once, but as yet you will not be able to see the results:

```sql
SQL = "SELECT * FROM BOOK_AUTHORS"
* 1. allocate a structure for the statement. This identifies the statement
Ok = SQLAllocStmt(hConnect, hStat)

* Execute a direct statement e.g. CREATE
Ok = SQLExecDirect(hStat, SQL)

* (processing in here)

* Now free the statement
Ok = SQLFreeStmt(hStat, SQL.DROP)
```

bcidemo2
Fetching Data

Data can be fetched from a data source by use of an SQL SELECT statement. The precise syntax of the SELECT statement will vary depending on the capabilities and idiosyncrasies of the data source you are querying: a SELECT statement for a Microsoft Access database may not match the SQL*Plus syntax used to query an Oracle database.

SQL SELECT statements return data in the form of a record set: a two dimensional table of rows and columns similar to the output from a RetrieVe or UniQuery command. To read the data you must iterate through this record set row by row, setting variables to the various column contents each time.

Once the statement has executed the BCI knows how many columns of data to expect. This is reported using the SQLNumResultCols() function, which takes the statement handle as an argument and populates a BASIC variable thus:

Ok = SQLNumResultCols(hStat, NumColumns)

This can be added to the example program as below:

```basi
SQL = "SELECT * FROM BOOK_AUTHOR"  
* 1. allocate a structure for the statement. This indentifies the statement  
   Ok = SQLAllocStmt(hConnect, hStat)  
   
* Execute a direct statement e.g. CREATE  
   Ok = SQLExecDirect(hStat, SQL)  
   
* See how many columns are returned  
   Ok = SQLNumResultCols(hStat, NumColumns)  
   
Crt "The select statement returned ":NumColumns:" columns."  
* Now free the statement  
   Ok = SQLFreeStmt(hStat, SQL.DROP)
```

Once the number of columns is known, you can begin to fetch the data.

This is where the syntax gets slightly unusual.
Rather than requiring the developer to explicitly fetch each column in turn, each time the next row in the record set is retrieved; BCI first requires that you set up a mapping between the record set columns and a series of BASIC variables that will receive the column data. This may seem odd, but in practice it allows BCI to populate those variables as you work through the record set behind the scenes, using code that is more efficient than asking for each column individually using a separate function call into BCI. The result is, of course, quicker operation.

The column mapping is set up using the `SQLBindCol()` function. This binds a numbered column in the record set to a BASIC variable. The columns are numbered from one. In the example below, the record set columns are bound to elements of a static (DIMensioned) BASIC array:

```
DIM COLS(NumColumns)

For I = 1 To NumColumns
  * bind these to local variables
  Ok = SQLBindCol(hStat, I, SQL.B.DEFAULT, COLS(I))
Next
```

If the binding is successful, the data can then be fetched by iterating through the record set a row at a time. Each time a row is fetched, the variables identified in the `SQLBindCol` function are automatically populated with the column data for that row, using the `SQLFetch()` function:

```
Loop
  Ok = SQLFetch(hStat)
Until Ok = SQL.NO.DATA.FOUND Do
  For I = 1 To NumColumns
    Crt COLS(I):"|":
  Next
  Crt
Repeat
```
Describing the Results

The SQLFetch function provides access to the data in the record set, but how do you identify the various columns? BCI provides two useful functions to return information about a record set column: the `SQLDescribeCol` function and the `SQLColAttributes` function.

The `SQLDescribeCol` function returns the name and data type for a column in the record set:

```plaintext
Ok = SQLDescribeCol(hStat, ColNumber, ColumnName, DataType, Precision, Scale, Null)
```

The `SQLColAttributes` sets either the TextResult or IntResult variables depending on the specific attribute being requested:

```plaintext
Ok = SQLColAttributes(hStat, ColNumber, Attribute, TextResult, IntResult)
```
Using Parameter Queries

Repeated SQL statements, such as repeated selections or insertions, are more usually executed as parameterized queries. This technique issues an SQL statement as a template with actual values replaced by placeholders. When the developer wishes to run the query, the placeholders can be substituted for real values before execution.

Using parameter queries is far more efficient for repeated statements, as the database can prepare the query beforehand. When a query is run, the database must generate an 'execution plan' of how the query should be processed: this might specify the columns or expressions to evaluate, the tables to join, the use of indices. Using a parameter query allows the database to generate that execution plan once, and to then reuse the plan for subsequent executions, reducing the amount of work required and thereby improving performance.

Again, the ability to prepare queries and the format for substitution are dependent on the data source. For an Access database (any many other data sources) a question mark can act as a placeholder:

```
SELECT BOOK_TITLES.*, BOOK_AUTHORS.FULLNAME
FROM BOOK_TITLES LEFT JOIN BOOK_AUTHORS
ON (BOOK_TITLES.AUTHOR_ID = BOOK_AUTHORS.AUTHOR_ID)
WHERE BOOK_TITLES.AUTHOR_ID = ?
```

To prepare such a statement so that repeated requests can be made, BCI provides the `SQLPrepare` function. The `SQLPrepare` function prepares a SQL statement:

```
Ok = SQLPrepare(hStat, SQL)
```

Before the statement can be run, any placeholders are substituted with real values. This is performed using the `SQLBindParameter` function. This must specify not only the value, but the data type and scale of the data being applied:

```
Ok = SQLBindParameter(hStat, ColNumber, SQL.B.BASIC, SQL.CHAR, Size, Scale, Value)
```
The completed statement can then be executed using the **SQLExecute** function. This executes the prepared statement by supplying the statement handle. You cannot use the SQLExecuteDirect with a prepared statement:

\[ \text{Ok} = \text{SQLExecute}(\text{hStat}) \]

Putting these together provides a performant means of accessing data:

```
SQL = "SELECT BOOK_TITLES.*, BOOK_AUTHORS.FULLNAME "
SQL := "FROM BOOK_TITLES LEFT JOIN BOOK_AUTHORS "
SQL := "ON (BOOK_TITLES.AUTHOR_ID = BOOK_AUTHORS.AUTHOR_ID) "
SQL := "WHERE BOOK_TITLES.AUTHOR_ID = ?"

* 1. Allocate a statement environment and attach it to the open connection
   \[ \text{Ok} = \text{SQLAllocStmt}(\text{hConnect}, \text{hStat}) \]

* prepare the statement
   \[ \text{Ok} = \text{SQLPrepare}(\text{hStat}, \text{SQL}) \]

* Now it can be run
   Loop
      Crt "Author Number : ":
      Input AuthorID
      Until AuthorID = "" Do

* substitute the value
   \[ \text{Ok} = \text{SQLBindParameter}(\text{hStat}, 1, \text{SQL.B.BASIC}, \text{SQL.INTEGER, 0, 0, AuthorID}) \]

* and execute the statement
   \[ \text{Ok} = \text{SQLExecute}(\text{hStat}) \]

   (lines removed)
   \[ \text{Ok} = \text{SQLFreeStmt}(\text{hStat}, \text{SQL.CLOSE}) \]
   Repeat
```

bcidemo7/bcidemo7a
Updating Data

BCI is a two way street. A developer can update the remote database (assuming appropriate privileges are in place) by issuing SQL INSERT and UPDATE statements. The same placeholder structure should be used when posting data:

INSERT INTO BOOK_AUTHORS (AUTHOR_ID, FULLNAME, SURNAME) VALUES(?,?,?)

The INSERT and UPDATE statements do not generate a record set.

```sql
SQL = "INSERT INTO BOOK_AUTHORS(AUTHOR_ID, FULLNAME, SURNAME)
VALUES(?,?,?)"
Ok = SQLAllocStmt(hConnect, hStat)
Ok = SQLPrepare(hStat, SQL)
Loop
   Crt "New Author Number : ":; Input AuthorID
Until AuthorID = "" Do
   Crt "New Full Name : ":; Input FullName
   Crt "New Surname : ":; Input Surname
* substitute the values
   Ok = SQLBindParameter(hStat, 1, SQL.B.BASIC, SQL.INTEGER,
                          0, 0, AuthorID)
   Ok = SQLBindParameter(hStat, 2, SQL.B.BASIC, SQL.VARCHAR,
                          0, 0, FullName)
   Ok = SQLBindParameter(hStat, 3, SQL.B.BASIC, SQL.VARCHAR,
                          0, 0, Surname)
* and execute the statement
   Ok = SQLExecute(hStat)
Repeat
```

Reporting Errors

All of the BCI functions return an error status, or zero if no error has been encountered. To report more detailed information in the case of an error being raised, BCI provides the SQLError function.

This returns the error status for the BCI environment, a specific connection or a specific statement:

```sql
Ok = SQLError(hEnv, hConnect, hStat, BCIError, DBMSError, Text)
```
The BCIError returns a code representing an error raised by the BCI itself. If the error has been raised by the data source, for example in the case of a data type mismatch, this is returned in the DBMSError argument. These errors are codes: the full description of the error is returned in the Text.

```vba
Ok = SQLExecute( hStat)
If Ok <> 0 Then
  Ok = SQLError( hEnv, hConnect, hStat, BCIError, DBMSError, ErrorText)
  Crt "An error has occurred:"
  Crt BCIError: " - " : DBMSError : " [ " : ErrorText : "]"
End Else
  Crt "Changes applied successfully."
End
```

bcidemo9
Using the CONNECT command

BCI ships with a useful command interface named CONNECT. This acts as a general BCI client, allowing you to issue commands and view results.

CONNECT is supported on both UniData and UniVerse platforms. To invoke the command use the syntax:

**CONNECT datasource**

CONNECT prompts for a user name and password (if not required you can <return> across these) and displays a prompt from which you can issue SQL commands. If the command fails, CONNECT will display the ODBC error message, which can be helpful in testing your configuration and the syntax for any specific commands, before you come to encapsulate them in your code.

```
>CONNECT WYCHBOOKS
Enter username for connecting to 'WYCHBOOKS' DBMS [demo]:
Enter password for demo:
WYCHBOOKS> SELECT DISTINCT GENRE FROM BOOK_TITLES;
GENRE  
--------------------------------------------------
BIographies  
BusIness  
ClAssic  
CrIme  
DrAmA  
FaNtaSy  
FiCtion  
HiStOry  
HumOur  
LaNguAge  
10 rows selected
WYCHBOOKS> Q
```

Using the CONNECT command with the WYCHBOOKS data source.
About the Author

Brian Leach is an independent MultiValue consultant specializing in the U2 platforms and a former IBM Information Champion.

Discover more tutorials, guides and tools at:

http://www.brianleach.co.uk