libDatabase

What Is It?

libDatabase is a high level database abstraction library written in Transcript. It is built on top of RevDB and the Valentina external.

RevDB provides a minimal amount of database abstraction such as handlers for connecting to, executing queries against and managing cursors for various databases. libDatabase simplifies using databases in your Revolution applications by:

- Convert SQL calls into arrays
- Automatically creates INSERT and UPDATE SQL from Transcript arrays
- Create SQL dumps for backup
- Exports from one database format to another (i.e. MySQL to SQLite)

libDatabase allows you to spend less time coding your database calls and more time focusing on incorporating the data into your application.

Changes From Version 1

If you used version 1 of libDatabase there are a couple of changes you should be aware of.

- Errors are now thrown.
- libdb_getTableRow() and libdb_getTableRows() have been renamed to libdb_getTableRecord() and libdb_getTableRecords(). libdb_getTableRow(s) will still work but are mapped to libdb_getTableRecord(s)().
- libdb_getTableRecord() and libdb_getTableRecords() no longer return arrays. Instead, you pass an array as a parameter that will contain the results when the function completes. This reduces the amount of data that needs to be copied.
- The parameter order for libdb_updateTable has been modified. The array containing values to update now comes before the array which contains the search criteria.
- Functions which fetch or add/update data now take a parameter that allows you to specify encodings. See section below.
- Functions such as libdb_escape no longer take a database identifier as the first parameter but rather the database type such as “mysql” or “sqlite”.
- The schema for tableid has changed. The table is now defined with two fields- table_name and next_id. There is one entry from each table in your database.

Managing Database Connections

libDatabase can manage connections to multiple databases. To make a database available to the library you register the connection settings using libdb_registerDatabase(). Following are some examples of connecting to different databases:

MySQL

```
put “mysql” into tDBA[“Type”]
put “127.0.0.1” into tDBA[“Host”]
put “mydatabase” into tDBA[“Name”]
put “root” into tDBA[“Username”]
put “” into tDBA[“Password”]

libdb_registerDatabase “MySQLConnection”, tDBA
```

SQLite (using altSQLite)

```
put “sqlite” into tDBA[“Type”]
put “3” into tDBA[“Version”] --> FOR altSQLITE3
put “mysqlite.db” into tDBA[“FilePath”]
put “myemail@myemail.com” into tDBA[“Email”]
put “myserialnumber” into tDBA[“SerialNum”]

libdb_registerDatabase “SQLiteConnection”, tDBA
```

Valentina 2

```
## Register a local Valentina database
put “valentina” into tDBA[“Type”]
put 2 into tDBA[“Version”]
put theDatabaseFolder & “/myvalentina.vdb” into tDBA[“Name”]
put “MyMacSerialNum” into tDBA[“MacSerial”]
put “MyWinSerialNum” into tDBA[“WinSerial”]

libdb_registerDatabase “ValentinaConnection”, tDBA
```

The first parameter of libdb_registerDatabase is an identifier string of your choosing. You will pass this string to many of the libDatabase handlers so the library knows what connection settings to use. The second parameter is an array which has the necessary connection settings for the database you are connecting to.

Error Reporting

Before going on we will discuss how errors are handled in many of the libDatabase functions. Most handlers will “throw” errors meaning that you should wrap calls to the libDatabase handlers in try/catch statements. The error thrown will always start with “libdberr,” and then the error message. In general this will make your code easier to read and maintain when executing multiple database calls. Here is an example:

```
on viewGetRecords
      local tRecordA,tRowsA
      local tSQL1,tSQL2,e

      try
            --> CREATE tSQL1 AND tSQL2 HERE...
```

```
            get libdb_getTableRecord("MyConnection", tRecordA, tSQL1)
            get libdb_getTableRecords("MyConnection", tRowsA, tSQL2)

            --> MORE CODE HERE
      catch e
            if item 1 of e = "libdberr" then
                  delete item 1 of e
                  answer "Database error:"&& e
            else
                  answer "An error occurred:" && e
            end if
      end try
end viewGetRecords
```

Error handling is not included in the examples contained in this article for the sake of brevity.

Executing SQL

To execute SQL commands that don't return record sets use libdb_executeSQL. Pass the database identifier and the SQL call. The returned value is the number of affected rows.

```
put libdb_executeSQL("MyConnection", tSQL) into tAffectedRows
```

Fetching Records

There are three functions you can use to fetch records using libDatabase. They are:

- libdb_getTableRecord()
- libdb_getTableRecords()
- libdb_dataFromSQL()

libdb_getTableRecord() and libdb_getTableRecords both populate arrays. libdb_data-FromSQL() is similar to revDataFromQuery() in that it returns data using a column and line delimiter that you specify. (see Transcript dictionary).

```
--> PUT THE RESULTS INTO THE tData VARIABLE.  Each column will be
--> SEPARATED BY TAB, EACH LINE BY CR
get libdb_dataFromSQL("MyConnection", tData, tSQL, tab, cr)
```

libdb_getTableRecord() populates a single dimensional array since it is used to fetch a single record form a SQL query.

```
local tDataA
```

```
put “SELECT ID, Name, Address FROM People WHERE ID = 1” into tSQL
get libdb_getTableRecord(“MyConnection”, tDataA, tSQL)

put tDataA[“ID”] into field “ID”
put tDataA[“Name”] into field “Name”
put tDataA[“Address”] into field “Address”
```

libdb_getTableRecords() populates what we will refer to as a Data Array. This array contains a “Length” key which tells you how many records were returned. Each record that was returned is stored in the “Data” key and is accessed sequentially.

```
local tDataA

put “SELECT ID, Name, Address FROM People” into tSQL
get libdb_getTableRecords(“MyConnection”, tDataA, tSQL)

repeat with i = 1 to tDataA[“Length”]
      put tDataA[“Data”,i,”Name”] &tab& tDataA[“Data”,i,”ID”] &cr after tText
end repeat
delete last char of tText

set the text of field “SelectionMenu” to tText
```

Updating Records

You can update records using libDatabase without writing any SQL. libdb_updateTable handles all of this for you. This includes escaping and quoting the text you are inserting into the database. Here is a sample of how an update is done:

```
put the text of field “Name” into tUpdateA[“Name”]
put the text of field “Description” into tUpdateA[“Description”]

put the text of field “ID” into tWhereA[“ID”]

get libdb_updateTable(“MyConnection”, “store_items”, tUpdateA, tWhereA)
```

libdb_updateTable is defined as follows:

```
function libdb_updateTable pTargetDB, pTable, @pFieldValuesA, @pSearchA, pEn-
codingsA
```

pTargetDB - the connection identifier.

pTable - the name of the table to update.

pFieldValuesA - an array whose keys match fields in the table. The values of each key will be used to update the field in the table. You do not need to escape or quote the array values. This is handled automatically by the library.

pSearchA - an array used to find the record to update. Each key should match a field in the table. The value will be used to generate the WHERE clause of the SQL statement. Any string value will be quoted and escaped for you. If you want to update multiple records using an IN statement just separate the different values with commas. For instance, to update multiple record ids you could create an update array like this:

```
put “1,5,8,9” into tWhereA[“ID”]
```

pEncodingsA (not shown in example) - an array used to specify the type of encoding to apply to certain fields before inserting. See section on Encodings.

Adding Records

Adding records is similar to updating records. You don’t need to write any SQL. You just put the values for each field into an array. There are two calls for adding a record to a table - libdb_addToTable() and libdb_addNextToTable().

The parameters for the two functions are similar to libdb_updateTable.

libdb_addToTable

libdb_addToTable is defined as follows:

```
function libdb_addToTable pTargetDB, pTable, @pFieldValuesA, pTableField,
pEncodingsA
```

pTargetDB, pTable, pFieldValuesA and pEncodingsA are exactly the same as for libdb_updateTable. The fourth parameter, pTableField, is the name of one field in the table that the record is being added to. This is used internally to get a description of the table from the database.

libdb_addNextToTable

libdb_addNextToTable can be used when you are defining your own unique keys for records in your tables (i.e. you aren’t using an autoincrement field in a table for MySQL). If you are moving your database between different database vendors this may be necessary.

The way this works is that you define a table in your database which has the next unique integer for each of the other tables in your database. If my database has three tables named cars, boats and houses then I would create a fourth table that looked like this:

```
CREATE TABLE tableid (table_name varchar(40), next_id integer);
```

You would then create an entry for each of the three tables:

```
INSERT INTO tableid (table_name, next_id) VALUES(‘cars’,1)
INSERT INTO tableid (table_name, next_id) VALUES(‘boats’,1)
INSERT INTO tableid (table_name, next_id) VALUES(‘houses’,1)
```

libDatabase has a function named libdb_getNextTableID() which retrieves the next id and increments next_id by one for a table that has the structure as defined above. libdb_addNextToTable combines libdb_addToTable and libdb_getNextTableID() into one function. It is defined as follows:

```
function libdb_addNextToTable pTargetDB, pTable, @pFieldValuesA, pIDField,
pIDTable, pEncodingsA
```

pTargetDB, pTable, pFieldValuesA and pEncodingsA are exactly the same as for libdb_updateTable. pIDField is the name of the field that uniquely identifies each record in the table (i.e. ID). pIDTable is the name of the table that has the unique values. In the example above this would be “tableid”.

Note that libDatabase provides very basic ID numbers in that it does not try to reuse IDs. It just increments the ID by 1 each time a record is added to the database.

Using Cursors

For large data sets putting the results into an array may not be wise. In this case you can use libDatabase to work with cursors directly. Please see libDatabase.html for current documentation.

libdb_openCursor()
libdb_closeCursor
libdb_cursorRecordCount()
libdb_getCursorRecord()
libdb_nextCursorRecord()
libdb_previousCursorRecord()
libdb_cursorHasRecords()

SQL Dumps

libDatabase allows you to create SQL dumps without using any additional programs. The libdb_getSQLDump() function will return a dump of the given database. You can choose whether to include DROP, CREATE and/or INSERT statements. To return a dump of all database tables and their contents you would call:

```
get libdb_getSQLDump(“MyConnection”, empty, true, true, true, empty, empty)
```

If you wanted only the DROP/CREATE statements for certain tables you would call:

```
get libdb_getSQLDump(“MyConnection”, “MyTable1,MyTable2”, true, true, false,
empty, empty)
```

The examples above will return the SQL. If you want to save the SQL to a file you can provide a full path to the file as the last parameter. This method uses less memory since the sql dump is written to the file as it is created rather than stored in memory.

```
get libdb_getSQLDump("MyConnection", empty, true, true, true, empty, "C:/
mySQLDump.sql")
```

Converting From One Database to Another

libDatabase allows you to export one database to another. For example, you may develop a project using a MySQL database but then deliver to the customer using a local database such as SQLite or Valentina. The libdb_exportDB handler makes this easy.

libdb_exportDB requires at least two parameters - the source database identifier and the target database identifier. The source database identifier must be a valid database that has data in it. The target database identifier contains the connection information for a database that may not exist yet. Here is how you would export a MySQL database to Valentina:

```
--> REGISTER MYSQL DB
put "mysql" into tDBA["Type"]
put "127.0.0.1" into tDBA["Host"]
put "mydatabase" into tDBA["Name"]
put "root" into tDBA["Username"]
put "" into tDBA["Password"]

libdb_registerDatabase "MySQLConnection", tDBA

--> REGISTER SQLITE DB
put "sqlite" into tDBA["Type"]
put "3" into tDBA["Version"] --> FOR altSQLITE 3
put "mysqlite.db" into tDBA["FilePath"]
put "myemail@myemail.com" into tDBA["Email"]
put "myserialnumber" into tDBA["SerialNum"]

libdb_registerDatabase "SQLiteConnection", tDBA

--> EXPORT MYSQL TO SQLITE
libdb_exportDB "MySQLConnection", "SQLiteConnection"
```

The above code would create a new SQLite database and insert the tables and data contained in the MySQL database.

For information on additional parameters you can pass to libdb_exportDB please see the library documentation.

Encodings

When fetching and inserting data into databases from Revolution you usually need to deal with different text encodings. It is a good idea to decide which encoding you want to store your data in and then make sure that all data being inserted into the database is

encoded properly. If all data is stored using the same encoding then you can determine when data needs to be encoded differently when displaying on a certain platform.

For example, when using MySQL you should convert text entered on a Macintosh (mac-roman) to its ISO equivalent using macToISO when inserting into the database. Then whenever you fetch data from MySQL while running on Macintosh you can apply ISOtoMac. This avoids certain characters appearing differently on different platforms.

Furthermore, when dealing with Unicode text you will often need to convert between UTF8 (which is used in most databases) to UTF16 (which is used in Rev fields).

libDatabase tries to make this a little easier by allowing you to define these mappings using the pEncodingsA array parameter. The keys of pEncodingsA match the names of fields in the SQL query. For example, if you have the SQL statement

```
SELECT ID, Name, Title FROM employees
```

and you want to convert Name and Title from UTF8 to UTF16 you would do the following:

```
local tDataA

put “UTF8toUTF16” into tEncodeA[“Name”]
put “UTF8toUTF16” into tEncodeA[“Title”]

put “SELECT ID, Name, Title FROM employees WHERE ID = 1” into tSQL

get libdb_getTableRecord(“MyConnection”, tDataA, tSQL, tEncodeA)

set unicodeText of fld “Name” to tDataA[“Name”]
set unicodeText of fld “Title” to tDataA[“Title”]
```

The same goes for updating or adding records. Following is an example of how to ensure that text entered on a Mac is always encoded as ISO when inserting into a database.

```
put text of fld “ID” into tWhereA[“ID”]

put text of fld “Name” into tUpdateA[“Name”]
put text of fld “Description” into tUpdateA[“Description”]

put “ISOtoMac” into tEncodeA[“Name”]
put “ISOtoMac” into tEncodeA[“Description”]

get libdb_updateTable(“MyConnection”, “myTable”, tUpdateA, tWhereA, tEncodeA)
```

Possible encoding values are:

- ISOtoMac - Use when data is stored as ISO in the database. Converts to/from ISO when running on a Mac. Not applicable to Valentina 1.x databases since Valentina already modifies encodings as necessary.
- ISOtoUTF16 - Use when data is stored as ISO in the database but you want to use unicode text in Revolution. Used only with libdb_getTableRecord and libdb_getTableRecords.
- ISOtoUTF8 - Use when data is stored as UTF8 in the database but you want to use ISO text in Revolution.
- UTF8toUTF16 - Use when database stores text as UTF8. Converts to/from UTF16 for use in Revolution.