GM8SQLite3CPP

Overview

Out of the box, Game Maker version 8 (hereafter referred to as GM8) has a means to read and write data structures between disk and memory. For example, there are functions to write maps, lists, etc.. However, GM8 does not deliver a query engine. Nor does it offer a means of ensuring referential integrity between data records. Many of the features lacking in GM8's record handling model are available, out of the box, with just about any database that supports the SQL language. So GM8 would benefit greatly from an SQL interface to such a database. This would speed up development a great deal since most developers (professional and otherwise) have a working knowledge of the SQL language and would not have to write their own query engine with GM scripting. However, most databases supporting an SQL interface are rather heavy in terms of footprint as well as learning curve. So requiring a client to deploy something even as small as MySQL is a bit of overkill for most PC games. Here is where SQLite comes into play.

> "SQLite is a software library that implements a self-contained, server-less, zero-configuration, transactional, SQL database engine. SQLite is the most widely deployed SQL database engine in the world." (www.sqlite.org, 6/6/2013).

SQLite basically provides 2 interfaces.

1. An executable that you can run on a Windows machine to create and manage a file based database.

2. A code library developers can build into their applications interact with databases. The code library is written in C but it exposes functions that allow the program user to manage schema and data using the SQL language.

SQLite provides many of the robust features of larger, enterprise databases without the overhead. A perfect fit for most PC and even medium sized server based games and thus perfect for GM8. However, in order for GM8 to access the C library, an extension (containing a .dll file) is needed. This extension wraps the SQLite engine and exposes SQLite features to GM8 as GML functions. GM8SQLite3CPP is such an extension. The rest of this document provides the information needed to use GM8SQLite3CPP in GM8.

Version Compatibility
GM8SQLite3CPP was built using C++ and, as you can guess from the name, was designed and tested against GM version 8 and SQLite version 3 (more precisely 3.7.1.7). However, that's not to say it won't work in other versions. You would just have to test it and see. You should also know that the dll at the heart of the extension was created with Microsoft Visual Studio 2010 Express and this created a dependency for C++ Redistributable Package 2010 on the client running GM8. If you don't have this installed on the PC, it will give you an error at run-time stating that MSVCR100.dll is missing. If this is a big problem, I, or you, if you want to use the source, can compile it to use an older C++ package.

Project Driver
I believe there is another GM extension out there that provides an interface to SQLite. However, that version did not appear to offer foreign key constraint or extended result codes. So that was the driving force behind me writing GM8SQLite3CPP. Plus the fact that I had never written a GM extension or DLL and wanted to see how it was done ☺. I believe the other extension had a 5 connection and 5 query/statement limit. My extension has no limit on the number of connections per database or statements per connection.

Disclaimer
I am not a C++guru or professional so I can't vouch for the efficiency or this extension. If you use it and want to see something done differently, you are welcome to the source code or can contact me with a request and I might consider enhancing it. Just email rob.shoults@gmail.com.

Basic Usage
1. Create a connection object using function sql3_connection_create or sql3_connection_adv_create. This function will give you a connection id that you will use to interact with the connection. You can connect to an existing SQLite3 database file, create a new one, or even specify that you want a connection to a new database in memory.
2. Create statements containing SQL to interact with your data base through the connection object. Use the sql3_statement_create function to create a statement. This function will give you a statement id that you will use to interact with the statement.
3. Execute a statement using the sql3_statement_step function. This will produce a result code based on the type of SQL statement you executed. You evaluate the result code to determine your next move. For example, if you execute a statement that is not designed to return data, then you get a result code simply indicating if the statement executed successfully. As an example, let's say you just added a record using INSERT INTO and it created a new record in a table. You would get a result code back indicating the statement was done (code -101). Let's say that your table was designed to auto-

generate a rowid as the primary key. You could retrieve this newly created key using sql3_connection_last_insert_rowid_get. If your statement was intended to produce a result set (i.e. return data), then you can retrieve an entire record in the result set as a string using the function sql3_statement_row_data_get. After retrieving a row, you can move to the next row by calling sql3_statement_step again. You repeat this cycle until sql3_statement_step returns a code indicating it has reached the end of the result set (the done code of -101). If, at any time while stepping through a statement you need to get the column names available in the result set, you can call sql3_statement_row_column_names_get.

4. If you want to rerun a statement again after it has reached a done state, you can call function sql3_statement_reset. Then call sql3_statement_step to execute again.
5. When you are done with a statement, you remove it using the function sql3_statement_remove. This frees up memory and invalidates the statement id you received when you called sql3_statement_create.
6. When you are done with a connection, you remove it using the function sql3_connection_remove. This frees up memory and invalidates the connection id you received when you called sql3_connection_create or sql3_connection_adv_create. If your database was a file and this was the only connection to that file, then this function will also unlock the file at the operating system level. Before removing itself, the connection will remove all statements assigned to it.

Foreign Key Support

Your connections are setup to enforce foreign key constraint by default. You can toggle this on/off for a connection using the function sql3_connection_constraint_fkey_toggle.

Function Return Values

GM limits .dll function call return types to string or real. In the .dll, real translates to a double (decimal) type. Every GM8SQLite3CPP function returns something. Strings need no explanation but you should know that all GM8SQLite3CPP real returns do not contain decimal values so GM will strip the .0 allowing you to treat the return like an integer in your comparisons. So basically just treat real returns as integers.
Concerning return types, there are basically 3 types of functions in GM8SQLite3CPP.

1. A function that returns an id or a result code indicating why it could not return an id. Ids will always be positive integers and result codes will always be negative integers. An example function is be sql3_connection_create.
2. A function that returns a success status or a result code indicating why it could not complete an operation. Success will always be a positive 1 and result codes will always be a negative integer. An example function is sql3_connection_constraint_fkey_toggle.
3. A result code indicating one of many states of an object or operation. For example, calling sql3_statement_step can return multiple state indicators. Result codes returned from these functions will always be negative integers.

Example Returns

1. Example 1: I make a call to a function where I am expecting an id. If I get a positive integer returned, I know that is an id. If I get a negative integer returned, I know there was a problem. I immediately call another function that can interpret the negative integer and give me an English description of the problem.
2. Example 2: I make a call to a function that I expect to complete and return a success status. I should expect a positive 1 for success. If I get a negative integer instead, I

know I have a problem and I immediately call another function that can interpret the negative integer and give me an English description of the problem.

3. Example 3: I call a function that I am expecting to return any number of states to me and I need to screen for the various states. This type of function will always return a negative result code for you to evaluate. You will quickly learn some common expected result codes. For example calling sql3_statement_step should give a -100 or -101 in most cases depending on the type of sql statement you used. If you get something other than this and want to know what it is you immediately call another function that can interpret the negative integer and give me an English description of the problem.
4. Example 4: I call a function and I get a zero (0) result. This should never happen. If this happens it means that GM is not getting to the function and something is probably broken with the extension or you didn't call it correctly.

Interpreting Result Codes

1. So how do you know what a result code means? From the information above, you already know that 1 means success. However, all other result codes are negative integers and need interpreted. To get an English description of each code, you should call one of the following functions immediately after receiving the result code (in other words, you must call it before executing any additional functions against that connection).
    a. Use sql3_result_code_text_get for a result code between -1 and -89999, inclusive. These are codes returned by the SQLite engine itself. For example, if you pass me an invalid sql expression, my wrapper will pass that off to the SQLite engine without checking it. SQLite engine will reject it and pass back a result code to my wrapper which will just pass it back to you.
    b. Use sql3_result_code_90k_text_get for a result code beyond -89999. These are my custom error codes output from my wrapper. For example, if you pass me an invalid connection id, my wrapper code will detect that before it ever gets to the SQLite engine code. The wrapper will spit a result code back at you explaining the problem.
2. If you check the SQLite web site, you find that SQLite result codes are all positive integers. I convert them to negative integers in GM8SQLite3CPP so that I can provide more functionality from a single return value. For example, I can return positive ids or a negative result code from the same call and it is easy for the GM8 programmer to interpret. SQLite also likes to return 0 as a success status. This is not good for GM8 because GM8 uses it as a default value for failed function calls. So a GM8 programmer would have trouble with a 0 because he/she wouldn't know if it meant success or failure. So I always use a positive 1 to represent success and I eliminate the use of 0 altogether. You either get a 1 or a negative result code.
3. You will also notice on the SQLite web site that early versions of SQLite provided a limited set of result codes and later versions open up more detailed result codes called "extended result codes". SQLite toggles this off for newly created connections by default. In contrast, GM8SQLite3CPP by default toggles this on for newly created connections. You can toggle this on/off using the function sql3_connection_extended_result_codes_toggle.

Functions

sql3_connection_create(filename:string)connection:real

Use this function to create a file based database or to access and existing database. The function takes a string filename and returns a positive integer connection id. Use the connection id to communicate with the database through most of the other functions. This is a wrapper for the SQLite3 function sqlite3_open. For details see http://www.sqlite.org/c3ref/open.html.

Sample
(Here we create a connection using the simple call)

```
c1=sql3_connection_create("dbfile1.db");
```

sql3_connection_adv_create(filename:string;OPEN_FLAG_CONST:real;zvfs_or_"NULL":string)connection:real
Use this function to create a database or to access and existing database. This function is a wrapper for the sqlite3_open_v2 function which provides rather robust control over the database and connection. For details, see http://www.sqlite.org/c3ref/open.html. I have provided the following constants for use with the second function parameter (the open flag). There are as follows:

1. **SQLITE_OPEN_READWRITE_or_SQLITE_OPEN_CREATE** = 6
2. **SQLITE_OPEN_READONLY** = 1
3. **SQLITE_OPEN_READWRITE** = 2

Sample
(Here we create a connection using the advanced call)

```
c1=sql3_connection_adv_create("dbfile1.db",
SQLITE_OPEN_READWRITE_or_SQLITE_OPEN_CREATE, "NULL");
```

sql3_connection_remove(connection:real)resultcode:real
Use this function to remove a connection from memory. It removes all statements before removing the connection. Removing a connection also invalidates the connection id. This function wraps the SQLite function sqlite3_close function. For details, see http://www.sqlite.org/c3ref/close.html. Note, that this function returns a positive 1 on success and a negative result code on failure.

Sample
(Here we create a connection and then remove it)

```
c1=sql3_connection_create("dbfile1.db");
rc=sql3_connection_remove(c1);
```

sql3_last_result_code_get(connection:real)resultcode:real
Use this function to get the last result code listed for a connection. Note that you must call this function immediately after the call you are trying to evaluate the result code for. You typically won't need this function because your last action against the connection will return the result code. However, I have provided it for convenience in case you find some use for it. This is a wrapper for the SQLite function sqlite3_errcode. For details, see http://www.sqlite.org/c3ref/errcode.html.

Sample
(Here we create a connection, then a statement, then execute the statement, then we check the last result code on the connection. In this sample, rc1 and rc2 are the same value. Because the sql was valid, they both return a -101 result code which equates to a completed status)

```
c1=sql3_connection_create("dbfile1.db");
stmt1=sql3_statement_create(c1,"create table t1(id INTEGER NOT
NULL PRIMARY KEY, f1 varchar(30));");
rc1=sql3_statement_step(c1,stmt1);
rc2=sql3_last_result_code_get(c1);
```

sql3_last_result_code_text_get(connection:real)resultcodetext:string
Use this function to get the plain English text describing a result code that is between -1 and -89999, inclusive. Note that you must call this function immediately after the call you are trying to evaluate the result code for. The reason for this is that it checks the connection object to see what the most recent code was and then returns the text for that code. You may find yourself using this function quite a bit during application development as you debug your GM8SQLite3CPP calls. It is a wrapper for the SQLite function sqlite3_errmsg. For more details, see http://www.sqlite.org/c3ref/errcode.html.

Sample
(Here we create a connection, then we try to create a statement but it fails due to an SQL syntax error. As a result stmt1 = -1. The GM developer then calls sql3_last_result_code_text_get to get the text for the -1 result code.)

```
c1=sql3_connection_create("dbfile1.db");
stmt1=sql3_statement_create(c1,"create tab t1(id INTEGER NOT NULL
PRIMARY KEY, f1 varchar(30));");
msg=sql3_last_result_code_text_get(c1);
```

sql3_result_code_90k_text_get(90kresultcode:real)90kresultcodetext:string
Use this function to get the plain English text describing a result code that is beyond -90000. Unlike the **sql3_last_result_code_text_get** function, this function takes a result code and simply tells you what it means. So timing is not an issue when calling it. You can call it at any time and pass in a result code and it will return the text describing that result code. You may find yourself using this function quite a bit during application development as you debug your GM8SQLite3CPP calls. This function is specific to GM8SQLite3CPP so it does not wrap an existing SQLite function.

Sample
(Here we create a connection, then we try to create a statement but it fails due to an SQL syntax error. As a result stmt1 = -1. However, the developer fails to check stmt1. When sql3_statement_step is called, the wrapper within GM8SQLite3CPP detects that stmt1 is not a valid statement in its' statement list for connection c1. So it returns a result code of - 90002. Because this code is < -89999, to determine what this code means, the developer must call sql3_ sql3_result_code_90k_text_get and pass in the result code as a positive integer (i.e. 90002). sql3_result_code_90k_text_get returns the string "not a valid statement for this connection.")

```
c1=sql3_connection_create("dbfile1.db");
stmt1=sql3_statement_create(c1,"create tab t1(id INTEGER NOT NULL
PRIMARY KEY, f1 varchar(30));");
```

```
rc1=sql3_statement_step(c1,stmt1);
msg= sql3_result_code_90k_text_get (rc1);
```

sql3_connection_constraint_fkey_toggle(connection:real,toggle:real)bool:real
Use this function to toggle fkey constaint on/off for a connection. This function is a wrapper for the SQLite function sqlite3_db_config(connection, SQLITE_DBCONFIG_ENABLE_FKEY, … Notice that by default, SQLite creates a connection that does not enforce fkey constraints. In contrast, GM8SQLite3CPP has fkey constraints enables by default on newly created connections. So if you don't want the fkey constraint enabled for your new GM8SQLite3CPP connection, you have to call this function to toggle it off.

Sample
(Here, we turn off fkey constraints for the onnection c1.
rc = 1 for success and –int for failure).

```
c1=sql3_connection_create("dbfile1.db");
rc = sql3_connection_constraint_fkey_toggle(c1,0);
```

sql3_connection_extended_result_codes_toggle(connection:real;toggle:real)bool:real
Use this function to toggle extended result codes on/off. This function is a wrapper for the SQLite function sqlite3_extended_result_codes. Notice that by default, SQLite has extended result codes disabled on newly created connections. In contrast, GM8SQLite3CPP has extended result codes enabled by default on newly created connections. So if you don't want extended result codes enabled for your new GM8SQLite3CPP connection, you have to call this function to toggle it off.

Sample
(Here, we turn off extended result codes for the onnection c1.
rc = 1 for success and –int for failure).

```
c1=sql3_connection_create("dbfile1.db");
sql3_connection_extended_result_codes_toggle(c1,0);
```

sql3_statement_create(connection:real;sqlstring:string)resultcode:real
Use this function to create a new statement on the passed connection. The function will return a positive integer statement id or a negative integer result code that needs to be evaluated. The SQL string you pass should end in a semi-colon. The new statement will compile the sql string on creation so it is ready for execution using the sql3_statement_step function. This function wraps the SQLite function sqlite3_prepare_v2. For more details, see http://www.sqlite.org/c3ref/prepare.html.

Sample
(Here we create a connection, then a statement on the connection. stmt1is a positive integer representing a statement id if the call succeeded. If not, a negative integer result code is returned.)

```
c1=sql3_connection_create("dbfile1.db");
stmt1=sql3_statement_create(c1,"create table t1(id INTEGER NOT
NULL PRIMARY KEY, f1 varchar(30));");
```

sql3_statement_step(connection:real;statement:real)resultcode:real
This function executes the statement. This function wraps the SQLite function sqlite3_step. For more information, see http://www.sqlite.org/c3ref/step.html . It can return any number of result codes depending on how the execution worked. So you typically need to follow this statement with an evaluation of the result code. I do not provide result code constants so you will have to look them up on the SQLite site. See http://www.sqlite.org/c3ref/c_abort.html for basic result codes. Extended result codes are not tabulated anywhere that I could find so you may just have to call **sql3_last_result_code_text_get** for any codes returned that are not on the site I listed. If you come up with a tabulated list and want it added to the documentation and possibly to the extension as constants, email them to me. The most common result codes you will look for are -101 = DONE, -100 = a row is available in the result set.

Sample
(Here we create a connection, then create a statement, then we execute the statement using the sql3_statement_step function. rc1 will always return a –integer result code which must be evaluated to see if the call was a success and to see what state the statement is in following the call.)

```
c1=sql3_connection_create("dbfile1.db");
stmt1=sql3_statement_create(c1,"create tab t1(id INTEGER NOT NULL
PRIMARY KEY, f1 varchar(30));");
rc1=sql3_statement_step(c1,stmt1);
```

sql3_connection_last_insert_rowid_get(connection:real)rowid:real
You sometimes need tables that have an auto-incrementing primary key. This is very easy to establish in SQLite and even functions as a default behavior in a lot of cases. When you have a table setup to auto-generate the primary key, your program needs a way to get that key value after you create the record. This is accomplished by calling the sql3_statement_step function to execute a statement with an INSERT sql string. Immediately following the step call, you call sql3_connection_last_insert_rowid to get the newly created primary key.

Sample
(Here we create a table that has an auto-incrementing primary key. We then insert a record into the table and finally, we call this function to return the newly created primary key value.)

```
c1=sql3_connection_create("dbfile1.db");
stmt1=sql3_statement_create(c1,"create table t1(id integer
primary key, f1 varchar(30));");
rc1=sql3_statement_step(c1,stmt1);
rc2=sql3_statement_remove(c1,stmt1);
stmt2=sql3_statement_create(c1,"insert into t1 (f1) values
('hello');");
rc3=sql3_statement_step(c1,stmt2);
show_message("last_rowid:
"+string(sql3_connection_last_insert_rowid_get(c1)));
```

sql3_statement_reset(connection:real;statement:real)resultcode:real
After a statement has been executed using **sql3_statement_step** to the point that it returns a -101 result code (representing a DONE state), it cannot be executed again unless it is reset. Use this function to reset it. This function wraps the SQLite function sqlite3_reset. For more

details, see http://www.sqlite.org/c3ref/reset.html. After calling this function, you can re-execute the statement using the function **sql3_statement_step**.

Sample
(Here we create a table with an auto-incrementing primary key. We then create a statement (stmt2) to add a record to this table. We execute the statement once to add the first record, then we reset it, then we execute the same statement again to add another record.)

```
c1=sql3_connection_create("dbfile1.db");
stmt1=sql3_statement_create(c1,"create table t1(id integer
primary key, f1 varchar(30));");
rc1=sql3_statement_step(c1,stmt1);
rc2=sql3_statement_remove(c1,stmt1);
stmt2=sql3_statement_create(c1,"INSERT INTO t1 (f1) VALUES
('hello');");
rc3=sql3_statement_step(c1,stmt2);
rc4=sql3_statement_reset(c1,stmt2);
rc5=sql3_statement_step(c1,stmt2);
```

sql3_statement_remove(connection:real;statement:real)resultcode:real
Use this function to remove a statement from memory which also invalidates the statement id. This function wraps the SQLite function sqlite3_finalize. For more detail, see http://www.sqlite.org/c3ref/finalize.html. Note, that this function returns a positive 1 on success and a negative result code on failure.

Sample
(Here we create statement to create a table in our database. We then remove the statement to free up memory).

```
c1=sql3_connection_create("dbfile1.db");
stmt1=sql3_statement_create(c1,"create table t1(id integer
primary key, f1 varchar(30));");
rc1=sql3_statement_step(c1,stmt1);
rc2=sql3_statement_remove(c1,stmt1);
```

sql3_statement_row_column_names_get(connection:real;statement:real)row:string
If your query returns a result set when executed via **sql3_statement_step**, you will receive a result code = -100. To retrieve a row from the result set, you must use **sql3_statement_step** to move the cursor to the row and then call sql3_statement_row_data_get to have the row data returned as a pipe-delimited string. If you need the column names for the result set, you can get them as a pipe-delimited string by calling this function.

Sample
(Here we create a table and populate it with a record. We then create a new statement to get call records from the table (stmt3). We then use a while loop and sql3_statement_step to iterate over the result set. For each row in the table, we call sql3_statement_row_data_get to retrieve a string representing the data in the row. For the first row, we also grab a string of column names and show them.)

```
c1=sql3_connection_create("dbfile1.db");
```

```
stmt1=sql3_statement_create(c1,"create table t1(id integer
primary key, f1 varchar(30));");
rc1=sql3_statement_step(c1,stmt1);
rc2=sql3_statement_remove(c1,stmt1);
stmt2=sql3_statement_create(c1,"insert into t1 (f1) values
('hello');");
rc3=sql3_statement_step(c1,stmt2);
rc4=sql3_statement_remove(c1,stmt2);
stmt3=sql3_statement_create(c1,"select * from t1;");
cnt=1;
while (sql3_statement_step(c1,stmt3)=-100) {
      if (cnt=1) {
      show_message(sql3_statement_row_column_names_get(c1,stmt3))
      ;}
      show_message(sql3_statement_row_data_get(c1,stmt3));
      cnt+=1;
}
rc4=sql3_statement_remove(c1,stmt3);
rc5=sql3_connection_remove(c1);
```

sql3_statement_row_data_get(connection:real;statement:real)row:string
If your query returns a result set when executed via **sql3_statement_step**, you will receive a result code = -100. To retrieve a row from the result set, you must use **sql3_statement_step** to move the cursor to the row and then call this function to have the row data returned as a pipe-delimited string.

Sample
(Here we create a table and populate it with a record. We then create a new statement to get call records from the table (stmt3). We then use a while loop and sql3_statement_step to iterate over the result set. For each row in the table, we call sql3_statement_row_data_get to retrieve a string representing the data in the row. For the first row, we also grab a string of column names and show them.)

```
c1=sql3_connection_create("dbfile1.db");
stmt1=sql3_statement_create(c1,"create table t1(id integer
primary key, f1 varchar(30));");
rc1=sql3_statement_step(c1,stmt1);
rc2=sql3_statement_remove(c1,stmt1);
stmt2=sql3_statement_create(c1,"insert into t1 (f1) values
('hello');");
rc3=sql3_statement_step(c1,stmt2);
rc4=sql3_statement_remove(c1,stmt2);
stmt3=sql3_statement_create(c1,"select * from t1;");
cnt=1;
while (sql3_statement_step(c1,stmt3)=-100) {
      if (cnt=1) {
      show_message(sql3_statement_row_column_names_get(c1,stmt3))
      ;}
      show_message(sql3_statement_row_data_get(c1,stmt3));
      cnt+=1;
}
rc4=sql3_statement_remove(c1,stmt3);
rc5=sql3_connection_remove(c1);
```