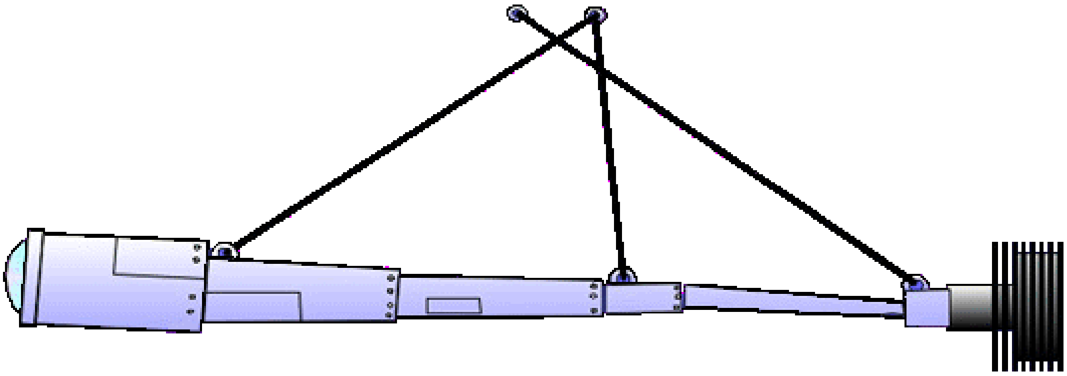
ForthQL User Manual

Easy SQL processing for VFX Forth for Linux



Rafael Gonzalez Fuentetaja

ForthQL
User manual
Manual revision 1.00
27 September 2008

Table of Contents

1 Introduction

1.1 Abstract

An implementation of **ForthQL** for *VFX Forh for Linux* is presented. SQL statements are embedded within words using `CHAR |` as separator. Dynamic SQL statements that bound parameters at runtime are also possible. Result sets are processed row by row by callback words. ForthQL relies on a simple DB API, where one connection (per thread) to the database is used. This is enough for most purposes. A sample implementation for *SQLite3* is also included. SQLite3 implementation includes a word set for non-callback processing style outside ForthQL.

1.2 References

1. *The Nearly Invisible database or ForthQL*, N. J. Nelson. 22th EuroForth Conference Proceedings, pp. 52-77, available at THE EUROFORTH WEBSITE.
2. *VFX Forth for Linux User Manual*. The evaluation license for Linux is free. Visit MPE Website for details.

2 Forth-SQL Interface

2.1 Introduction

This implementation of ForthQL depends on a two basic principles of operation:

- A simple API that hides the details of DBMS connections. Connection/Disconnection procedures are out of the scope of ForthQL.
- A Callback interface can be established with the DBMS to process the query result set row by row.

2.1.1 Portability

This module is written in *VFX Forth for Linux* but it can probably be ported easily to other Forth environments as well.

The following aspects of VFX has been used:

- Exception messages and codes.
- Modules, encapsulating words into modules and exporting words.
- GenIO Architecure. Memory Device.
- MPEisms. Convenience words defined by MPE for its products like `?comp`, `$CRLR`, `ALLOT&ERASE` or `?throw`

2.2 Embedding SQL code

Examples shown are based on the paper by N. J. Nelson cited in the `References` section.

The main idea of this interface is to embed multiline SQL statements inside colon definitions as shown below:

```
: TEST1
  SQL| INSERT INTO delegates      \ example comment 1
  VALUES ('Chuck','Moore',1) |SQL
;
```

SQL statements are started by the word `SQL|` and ended either with `|SQL`, `|SQL.` or `|SQL>` . They span one or more lines and Forth comments starting with `\` can appear in between. These comments are typically used to document the SQL code itself.

The SQL statement is compiled into a Z-string inside the current definition (`TEST1` in this case) and sent to the DBMS for execution when the current definition is executed. Apart from these comments, which are stripped in the actual SQL string, the syntax of the SQL statement depends on the actual DBMS being used.

Ending word variations `|SQL` or `|SQL>` are used whenever zero or one Forth callback action are used. See glossary section below.

Lack of SQL ending word inside a file will result in "delimiter not found" exception.

2.3 Dynamic SQL code

ForthQL allows not only static SQL statements to be embedded in Forth but also allows for dynamic SQL statement generation by means of SQL parameters as in the example:

```
: TEST2
  SQL| SELECT cuscode,cusname FROM customers  \ Fixed part
  WHERE cusid BETWEEN
  | LOWLIMIT |  AND | HIGHLIMIT               \ parameters
  |SQL> TESTOUT                               \ Output
;
```

In this example, we build a template SQL statement. Words `LOWLIMIT` and `HIGHLIMIT` are template parameters and will patch the SQL string with proper values at runtime, when `TEST2` is executed.

`CHAR |` is both used as the ending character of the static SQL string (as in the first and third appearance) and as a Forth word that restarts the SQL string compilation (as in the second appearance).

SQL parameters are specified like in the example below:

```
:noname    \ ca1 u1 --
  s"    12" drop -rot move ;
5 CHARS SQLParameter: LOWLIMIT

   s" 100009" drop -rot move ;
6 CHARS SQLParameter: HIGHLIMIT
```

At compile time, when the template SQL is being built, parameters need to specify how many space or 'width' they need in the SQL string. `LOWLIMIT` needs `5 CHARS` and `HIGHLIMIT` needs `6 CHARS`.

At runtime, SQL parameters run an execution token *xt* This token is for a word that must fill in just the part of the SQL string *ca1 u1* reserved to its parameter. Of course, *u1* is equal tothe parameter 'width'. Nameless definitons above will fill reserved spaces with literals `12` and `100009` respectvely.

This mechanism is totally independent from the usual SQL run time parameter binding using wildcards.

2.3.1 Parameter edition support

The above mechanisms are enough to edit any string needed into the parameter memory area. However, we can take advantage of VFX GenIO Architecture and use well known words like `TYPE`, `.` , `EMIT` and friends. The advantages are best appreciated when formatting numbers. The above example can be re-written as:

```
:noname                           \ ca1 u1 --
  [sqlio 12 . sqlio] ;
5 CHARS SQLParameter: LOWLIMIT

:noname                           \ ca1 u1 --
  [sqlio 100009 . sqlio] ;
8 CHARS SQLParameter: HIGHLIMIT
```

Words `[sqlio` opens a context where standard I/O is redirected to the *ca1 u1* memory region using the `SQL-MemDev` GenIO Device. Word `sqlio]` closes this device and restores standard I/O to previous values. Raw access to this memory zone is still possible using `IOCTL-GEN`s, but it should not be necessary in this approach.

Warnings:

- Take into account that word `.` emits a final space.
- Due to implementaion issues, reserve 1 additional `CHAR` to your planned parameter width.

2.4 Processing SQL output

Word `|SQL` just executes the SQL code for a statement that does not produce a *result set*, like an insertion into a table (as in `TEST1`). Of course, other situations will result in the retrieval of one or more rows in a result set, which can be processed one by one by a Forth word.

Word `|SQL>` - as shown in `TEST2` - will also compile some runtime code and the execution token for the next word (`TESTOUT` in this case). The runtime code is responsible to invoke `TESTOUT` for each row being retrieved. Some DBMS APIs like SQLite3 has a C callback interface designed for this purpose. See the proper section for details.

2.4.1 Support for OOP

Extending the concept of `|SQL>`, another word named `|SQL>>` lets you specify both a callback and a client data pointer. This gives some support to integrate callback actions with object methods.

This feature is highly dependant on the DBMS API and the OOP library used. The following example is done using SQLite3 and GForth 'objects.fs' library ported to VFX Forth for Linux.

```
sql-dry-run off

: quote   ( -- ) [char] ’ emit ;

: [qtype]   ( ca u -- )
   postpone quote
   postpone type
   postpone quote
; immediate

: creation
   SQL| CREATE TABLE IF NOT EXISTS Person(
      name TEXT PRIMARY KEY, surname TEXT, age INTEGER);
   |SQL.
;

: persons
   SQL| SELECT * FROM Person; |SQL.
;

object class
   cell% inst-var m-name
   cell% inst-var m-surname
   cell% inst-var m-age
end-class Person
```

```
Person methods
protected
  :noname                                \ ca1 u1 --
    [sqlio m-name $@ [qtype] sqlio]
  ; 16 chars SQLParameter: /name/

  :noname                                \ ca1 u1 --
    [sqlio m-surname $@ [qtype] sqlio]
  ; 16 chars SQLParameter: /surname/

  :noname                                \ ca1 u1 --
    [sqlio  m-age ? sqlio]
  ; 8 chars SQLParameter: /age/

  :m (refresh)  ( colValue** this -- )
    0 sql3-$@ evaluate m-age !
  ;m

  :noname   ( object* nCols colValue** colName** -- ior )
    drop nip swap (refresh)
    SQLITE_OK
  ; SQLite3Callback: <<refresh>>
```

```
public
   m:   ( ca1 u1 ca2 u2 age this -- ) \ overrides construct
    0 dup m-name ! m-surname !
    m-age ! m-surname $! m-name $!
  ;m overrides construct

  :m save-person   ( this -- )
     SQL| INSERT OR REPLACE INTO Person
     VALUES( | /name/ |  , | /surname/ | , | /age/ | );
     |SQL.
  ;m

  :m refresh ( this -- )
    ." Refresing age from database for " m-name $@ type cr
    SQL| SELECT age FROM Person WHERE name = | /name/
    |SQL>> this <<refresh>>
  ;m

end-methods

: Homer   s" Homer" ;
: Maggie  s" Maggie" ;
: Bart    s" Bart" ;
: Simpson s" Simpson" ;

Homer  Simpson 43 Person heap-new constant p1
Maggie Simpson 40 Person heap-new constant p2
Bart   Simpson 14 Person heap-new constant p3

s" simpson.db" db-open db-throw creation

p1 save-person
p2 save-person
p3 save-person
p1 refresh
```

2.5 Debugging SQL code

`VARIABLE SQL-DRY-RUN` controls the behaviour of `|SQL`, `|SQL>` and `|SQL.` If set, it will compile code to type the contents of the SQL buffer and then `EXIT`. Parameters are seen with its run-time values into their places. Memory overruns in the SQL buffer are a likely cause of SQL syntax errors.

This is a compilation flag. To disable it, you must turn it off and reload your code.

2.6 Tunning the SQL buffer size

SQL statements are compiled (as Z strings) inside the current definition in a similar way as words `S"` or `."` do. The size of this **per-word SQL Buffer** is is controlled by the VALUE `#SQLBuffer` with a default size. However, the user can change this value prior to defining a given SQL statement to tune for very large statements and to avoid waste of memory space for short statements.

A buffer too short for a given statement will result in a buffer overflow exception. Forth comments do not count against this limit.

Words `SQLBuffer>` and `>SQLBuffer`, used as a pair surronding the definition, change and restore to a previous value the SQL buffer size.

`VARIABLE SQL-TUNNING` controls the printing of compile-time diagnostics on the SQL buffer memory usage. By default it is turned off.

```
SQL-TUNNING ON
#148 CHARS SQLBuffer>
: MYDEF   ( -- )
  SQL| CREATE TABLE IF NOT EXISTS Person(
     oid INTEGER PRIMARY KEY,           \ object Id
     name TEXT,
     surname TEXT,
     age INTEGER
     );
     |SQL.
;
>SQLBuffer
\ and the output is ...
MYDEF SQL Buffer = 148 bytes , used = 148 , wasted = 0
```

Example above shows the exact amount of tweaking using `SQLBuffer>`, hence the 0 wasted bytes.

2.7 Multithreading

ForthQL depends on a simple API `DB-EXECUTE` or `DB-PROCESS` This API hides the database connections. It is up to the actual Forth DBMS driver to make this thread-safe For instance, it could declare a `USER` variable `DB-HANDLE`. to have several threads with its own connection.

Some data structures like `SQL-MemDev` are conditionally compiled into `USER` variables if `SQL-MULTITHREAD` is non-zero.

2.8 Limitations

- As stated elsewhere, this word set operates on one DB connection or at most one DB connection per thread.
- There could be **race conditions** if two threads execute **the same dynamic SQL query** like shown in `TEST2`, giving unpredictable results. Not only the SQL string buffer must should protected during edition itself at runtime, it should be locked until the SQL statement has been executed. Solving this problem transparently to the user for this unlikely situation adds a lot of complexity. The simplest workaround is to duplicate the SQL statement in two different words (i.e. `TEST2` and `TEST2'`) and let the threads execute each one of them.

- `SQL-RESULT` for SQLite3 is not as nice as presented in Nelson's paper. The Forth callback does not know when is the last time to be invoked and this prevents further clean-up actions.
- Defining a maximun field width for a `SQLParameter:` at compile time is not very flexible. Text like fields vary much in their contents and you must set an upper limit to them. Finding such limit is a compromise.

2.9 Glossary

2.9.1 Support words

General purpose words which could be placed elsewhere and not related to `ForthQL`. They may be placed in my *Extras* project in a future.

Caddr/len strings

`: STRING/ \ ca1 u1 u -- ca2 u2`
Get the string-matched *ca2 u2* from the string-remaining *ca1 u1* and the length *u* of the original string. (Jenny Brien).

`: /COMMENT                                       \ ca1 u1 -- ca2 u2`
Strip trailing Forth backlash comments from a string.

Input Specification and Parsing

`: PARSE-AREA@                \ -- ca u`
Get the as yet unparsed portion of the input buffer. (Jenny Brien).

`: PARSE-AREA!            \ ca u -- ;`
Set the portion of the input buffer still to be parsed to *ca u*. Must start within the input buffer! (Jenny Brien).

`: PARSE-AREA/                  \ ca1 u1 -- ca2 u2`
Get the already parsed string *ca2 u2* in the input buffer from the yet unparsed *ca1 u1* string. Similar to what `STRING/` does.

`: SKIP-CHAR                          \ ca1 u1 --`
Skip `1 CHARS` (usualy the `SCAN`ned character) from the remaining space *ca1 u1* in the Forth input buffer, updating the input buffer.

Memory operations

`: +MOVE                                   \ ca1 u1 ca2 u2 -- ca3 u3`
Move memory region *ca2 u2* to receiver memory buffer *ca1 u1*. Available receiver memory buffer is now *ca3 u3*.

2.9.2 Generic DB API

This simple API is needed for ForthQL. These words must be implemented by the underlying DBMS Forth driver module. You must include this driver module before `ForthQL` module. Connection to DBMS is out of the scope.

Usage of zero-teminated strings (Z-strings) are required to ease interfacing with foreign, C-based DBMS APIs.

`: db-execute                          \ z-addr -- ior`
Execute SQL statement *z-addr* not returning any output, like table creation or row insertion/update/delete. *ior* code signals operation result.

`: db-process                       \ z-addr xt -- ior`

Execute SQL statement *z-addr*. When complete, word given by *xt* is repetidely called, row by row, to process output. *ior* code signals operation result.

`: db-process-with                          \ z-addr xt1 xt2 -- ior`

Execute SQL statement *z-addr*. When complete, word given by *xt2* is repetidely called, row by row, to process output. *ior* code signals operation result. *xt1* is the execution token for a word that - when executed - return a cell with client data. Client data can be anything, but mostly will be an object handle for OOP support (the `this` keyword).

`<clientData> VALUE clientDataPtr   \ xt1 is ’ clientDataPtr.`

`: this ( -- handle) ... ;          \ xt1 is ’ this`

`: db-throw                        \ ior --`

Map DB specific *ior* into an appropiate user exception and throw it. Exceptions are DBMS specific.

`: db-print                        \ z-addr --`

Print the SQL statement *z-addr* being executed and its result set nicely formatted.

SQL Buffer management

`#256 CHARS Value #SQLBuffer`

SQL Memory Buffer default size in bytes.

`: SQLBuffer>                               \ n1 -- n2`

Set the current SQL buffer size to *n1* bytes. Return the old value *n2* for latter restoration. Intended to use in pair with `>SQLBuffer`. See example in the *Tunning the SQL buffer size* section.

`: >SQLBuffer                               \ n1 --`

Set the current SQL buffer size to *n1* bytes.

`Variable SQL-TUNNING`

Flag. If true, prints the actual size SQL string being compiled so that you can fine tune `#SQLBuffer` for that word.

`: ?sql-tunning                             \ ca1 u1 -- ca1`

Print a summary report of bytes being used for the SQL Buffer in the word being defined.

2.9.3 SQL Code Compiler words

`Variable SQL-DRY-RUN`

Activate a mode where only SQL compiler words only compile code to print the SQL code. Do not execute anything on the DBMS. **Warining:** This is a compile (loadtime) flag, not runtime option

Exceptions thrown at compile time.

`ErrDef SQLDelimErr "No SQL Delimiter | found"`

`ErrDef SQLBuffOvf  "SQL Buffer overflow"`

`: SQL|                                     \ -- ca1 u1`

Start SQL string compilation, initializing a SQL memory buffer and concatenating verbatim until following `CHAR |`. Any backlashed comments are removed. Preserves newline characters.

`: |                                        \ ca1 u1 - ca2 u2`

Resume SQL string compilation to SQL buffer *ca1 u1* until next `CHAR |` is found, as above. Leave remaining SQL buffer *ca2 u2*.

`: |SQL                        \ ca1 u1 --`

End current SQL statement compilation. Receives as input the remaining SQL buffer *ca1 u1* to find out and compile the SQL buffer starting address. Compile internaly a call to `DB-EXECUTE` with all necessary parameters. At run time, `DB-EXECUTE` is invoked, which will send the SQL statement for the DBMS to execute.

`: |SQL>                       \ ca1 u1 "action" --`

End current SQL statement compilation and compiles the following `"action"` word. Receives as input the remaining SQL buffer *ca1 u1* to find out and compile the SQL buffer starting address. Compile a call to `DB-PROCESS` with all necessary parameters. At run time, `DB-PROCESS` is invoked which will send the SQL statement for the DBMS to execute. For each row returned by the DBMS as the result, it will invoke the xt of `"action"`.

`: |SQL>>                              \ ca1 u1 "object" "action" --`

End current SQL statement compilation and compiles both the following `"object"` and `"action"` words. Receives as input the remaining SQL buffer *ca1 u1* to find out and compile the SQL buffer starting address. Compile a call to `DB-PROCESS-WITH` with all necessary parameters. At run time, `DB-PROCESS-WITH` is invoked which will send the SQL statement for the DBMS to execute. For each row returned by the DBMS as the result, it will invoke the xt of `"action"` with the xt of `"object"` as a parameter.

`: |SQL.                       \ ca1 u1 --`

End current SQL statement compilation. Receives as input the remaining SQL buffer *ca1 u1* to find out and compile the SQL buffer starting address. Compile a call to `DB-PRINT` into the current definition. At run-time, `DB-PRINT` is executed which sends the SQL statement to the DBMS for execution and prints (a more or less nicely formatted) result set.

2.9.4 Dynamic SQL Statements

`: SQLParameter:                \ xt width -- ; [child] ca1 u1 pfa -- ca2 u2`

Define a *SQL parameter* that will get substituted at runtime when the definition containing the SQL code is executed. *width* is the 'parameter width', that is the amouts of bytes in the SQL string that must be reserved to be patched later on. Children words receive the remaining SQL buffer *ca1 u1* to just to compile *ca1 width* into the word that is defining the SQL statement; along with the *xt* and to calculate the remaining buffer. *ca2 u2*. *xt* is the execution token for an action word that performs this patching, receiving exactly the given memory area, like:

`: parameter-filler   ( ca1 width -- ) ... ;`

Support to parameter edition

The following words help parameter edition at runtime by doing I/O at the memory area to fill. They are not strictly needed but they are quite convenient.

`TextBuff: SQL-MemDev`

GenIO Memory Device SID. The multithreaded version is a `USER` variable, that must be initialized per thread with the phrase `SQL-MemDev initTextBuffSid`.

`: +SQLMemDev                           \ ca1 u1 --`

Open the `SQL-MemDev` GenIO memory device to edit memory zone *ca1 u1*. Standard I/O is redirected to this device. Words like `EMIT`, `.` or `TYPE` write on this region.

`: -SQLMemDev        \ --`

Close the `SQL-MemDev` GenIO Memory Device to for parameter edition. **Warning:** Standard I/O is not yet restored to previous value.

`: [sqlio                               \ ca1 u1 -- R: -- ip-handle op-handle`

Convenience macro for the phrase `[io +SQLMemDev` . Open the `SQL-MemDev` memory device, configures it to use buffer *ca1 u1* and redirects standard I/O to it. Intended to use inside a colon definiton at the start of a SQLParameter action. See the `Dynamic SQL code` section at the beginning of this chapter.

```
: sqlio]                                 \ R: -- ip-handle op-handle
```

Convenience macro for the phrase `-SQLMemDev io]` . Close the `SQL-MemDev` memory device and restores standard I/O to its previous settings. Intended to use inside a colon definiton at the end of a SQLParameter action. See the `Dynamic SQL code` section at the beginning of this chapter.

3 SQLite3 Database Interface

3.1 Introduction

This module is a Forth DBMS driver to SQLite3. It implements a minimal interface for use with *ForthQL* and an extended interface for SQL3Lite specific operations.

3.1.1 Portability

This module is written in *VFX Forth for Linux* but it can probably be ported easily to other Forth environments as well.

The following aspects of VFX has been used:

- C Callback Mechanism.
- Exception messages and codes.
- Modules, encapsulating words into modules and exporting words.
- MPEisms. Convenience words defined by MPE for its products like `?comp`, `$CRLR`, `ALLOT&ERASE` or `?throw`

3.2 Glossary

`0 Constant sql-multithread`

Compilation option when including this module. All DBMS drivers should test & define this value.

`MODULE SQLITE3`

3.2.1 C interface

SQLite3 return codes

```
0 Constant SQLITE_OK         \ Successful result
1 Constant SQLITE_ERROR      \ SQL error or missing database
2 Constant SQLITE_INTERNAL   \ An internal logic error in SQLite
3 Constant SQLITE_PERM       \ Access permission denied
4 Constant SQLITE_ABORT      \ Callback routine requested an abort
5 Constant SQLITE_BUSY       \ The database file is locked
6 Constant SQLITE_LOCKED     \ A table in the database is locked
7 Constant SQLITE_NOMEM      \ A malloc() failed
8 Constant SQLITE_READONLY   \ Attempt to write a readonly database
9 Constant SQLITE_INTERRUPT  \ Operation terminated by sqlite_interrupt()
#10 Constant SQLITE_IOERR    \ Some kind of disk I/O error occurred
#11 Constant SQLITE_CORRUPT  \ The database disk image is malformed
#12 Constant SQLITE_NOTFOUND \ (Internal Only) Table or record not found
#13 Constant SQLITE_FULL     \ Insertion failed because database is full
#14 Constant SQLITE_CANTOPEN \ Unable to open the database file
#15 Constant SQLITE_PROTOCOL \ Database lock protocol error
#16 Constant SQLITE_EMPTY    \ (Internal Only) Database table is empty
#17 Constant SQLITE_SCHEMA   \ The database schema changed
#18 Constant SQLITE_TOOBIG   \ Too much data for one row of a table
#19 Constant SQLITE_CONSTRAINT      \ Abort due to contraint violation
#20 Constant SQLITE_MISMATCH \ Data type mismatch
```

```
#21 Constant SQLITE_MISUSE   \ Library used incorrectly
#22 Constant SQLITE_NOLFS    \ Uses OS features not supported on host
#23 Constant SQLITE_AUTH     \ Authorization denied

#100 Constant SQLITE_ROW     \ sqlite_step() has another row ready
#101 Constant SQLITE_DONE    \ sqlite_step() has finished executing
```

SQLite3 data types

```
1 Constant SQLITE_INTEGER
2 Constant SQLITE_FLOAT
3 Constant SQLITE_TEXT
4 Constant SQLITE_BLOB
5 Constant SQLITE_NULL
```

```
: .sql3-version
```

Print the SQLite3 version being loaded The SQLite3 version number is an integer with the value (X*1000000 + Y*1000 + Z).

3.2.2 Generic DB API

Exceptions

```
ErrDef SQL3ColumnIndex "Index out of bounds in column widths array"
```

```
ErrDef SQL3ColumnWidthError "Column specifier not within range"
```

The ones below comes from the SQlite3 DBMS itself.

```
ErrDef SQL3_ERROR      "SQLite3: SQL error or missing database"
ErrDef SQL3_INTERNAL   "SQLite3: An internal logic error in SQLite"
ErrDef SQL3_PERM       "SQLite3: Access permission denied"
ErrDef SQL3_ABORT      "SQLite3: Callback routine requested an abort"
ErrDef SQL3_BUSY       "SQLite3: The database file is locked"
ErrDef SQL3_LOCKED     "SQLite3: A table in the database is locked"
ErrDef SQL3_NOMEM      "SQLite3: A malloc() failed"
ErrDef SQL3_READONLY   "SQLite3: Attempt to write a readonly database"
ErrDef SQL3_INTERRUPT  "SQLite3: Operation terminated by sqlite_interrupt()"
ErrDef SQL3_IOERR      "SQLite3: Some kind of disk I/O error occurred"
ErrDef SQL3_CORRUPT    "SQLite3: The database disk image is malformed"
ErrDef SQL3_NOTFOUND   "SQLite3: (Internal Only) Table or record not found"
ErrDef SQL3_FULL       "SQLite3: Insertion failed because database is full"
ErrDef SQL3_CANTOPEN   "SQLite3: Unable to open the database file"
ErrDef SQL3_PROTOCOL   "SQLite3: Database lock protocol error"
ErrDef SQL3_EMPTY      "SQLite3: (Internal Only) Database table is empty"
ErrDef SQL3_SCHEMA     "SQLite3: The database schema changed"
ErrDef SQL3_TOOBIG     "SQLite3: Too much data for one row of a table"
ErrDef SQL3_CONSTRAINT "SQLite3: Abort due to contraint violation"
ErrDef SQL3_MISMATCH   "SQLite3: Data type mismatch"
ErrDef SQL3_MISUSE     "SQLite3: Library used incorrectly"
ErrDef SQL3_AUTH       "SQLite3: Authorization denied"
```

Functions

`: db-open                        \ ca1 u1 -- ior`

Open a connection to a SQLite3 database file given by path *ca1 u1*. Returned handle is stored in `SQL3-HANDLE` `USER` variable. Does nothing if already open (`SQL3-HANDLE` is non-zero).

`: db-close                       \ -- ior`

Close a connection to a SQLite3 database. Handle is taken from `SQL3-HANDLE`. Clears `SQL3-HANDLE` as well. Does nothing if already closed (`SQL3-HANDLE` is zero).

`: db-errmsg                      \ ior -- ca1 u1`

Get an error description message *ca1 u1* from an *ior*. **Warning:** In SQLite3, *ior* is not looked up, the most recent error produced is retrieved.

`: db-execute                     \ z-addr -- ior`

Execute SQL statement *z-addr* not returning any output.

`: db-process                     \ z-addr xt -- ior`

Execute SQL statement *z-addr*. When complete, word given by *xt* is repetidely called, row by row, to process output.

`: db-process-with                        \ z-addr xt1 xt2 -- ior`

Execute SQL statement *z-addr*. When complete, word given by *xt2* is repetidely called, row by row, to process output. *ior* code signals operation result. *xt1* is the execution token for a word that - when executed - return a cell with client data. Client data can be anything, but mostly will be an object handle for OOP support (the `this` keyword).

```
<clientData> VALUE clientDataPtr   \ xt1 is ’ clientDataPtr.
: this ( -- handle) ... ;          \ xt1 is ’ this
```

`: db-throw                               \ ior --`

Map SQLite3 *ior* into an appropiate VFX Forth exception and throw it.

`: db-print                               \ z-addr --`

Print the SQL statement *z-addr* being executed and its result set nicely formatted.

`: SQLite3Callback:                       \ xt --`

Declare a SQLite3 callback that, when called by the Linux C interface it will execute in turn the action whose execution token is *xt* The action Forth word must have a signature like its C counterpart:

```
int action(void* cliData, int nCols, char* colValue[], char* colName[]);
```

`: sql3-$@                     \ a-addr i -- ca u`

Retrieve a string *ca u* given the base array of pointers *a-addr* to Z-strings and the index into the array *i*. Intended to retrieve contents from the callback Forth word being used to process rows in the ForthQL. See `SQLite3Callback:`.

3.2.3 SQLite3 Specific DB API

This API export function for a non callback driven interface. Iteration on the *result set* is done by the application itself.

`: sql3-prepare                           \ z-addr1 -- z-addr2 ior`

Precompiles a SQL statement *z-addr1* for later execution through one or more calls to `SQL3-STEP`. Only a single SQL statement is compiled, remaining is left as *z-addr2* for subsequent calls to `SQL3-PREPARE`. An empty statement set `SQL3-STH` to zero.

`: sql3-step         \ -- ior`

Get next row from the result set. *ior* is `SQLITE_ROW` if result set not complete or `SQLITE_DONE` otherwise. A different value signals an error.

`: sql3-reset        \ -- ior`
Reset the execution of a SQL statement.

`: sql3-#cols       \ -- n`
Return the number of columns in the result set. Only works after calling `SQL3-PREPARE`.

`: sql3-finalize    \ -- ior`
Clean up the result set after being iterated. Las thing to do after a `SQL3-PREPARE` and zero or more `SQL3-STEP`s. *ior* signals an error condition.

`: sql3-NULL         \ -- ca1 u1`
The verbatim `NULL` string with its four characters.

`: sql3-col-name    \ n -- ca u`
Return the column name for column *n*. `NULL` datatype is retrieved as the `NULL` string.

`: sql3-col-text    \ n -- ca u`
Return the column value as ASCII text for column *n*. `NULL` datatype is retrieved as the `NULL` string.

`: sql3-col-type   \ n1 -- n2`
Return the column type for column *n1* as defined in one of the SQLITE constants `SQLITE_INTEGER` through `SQLITE_NULL`.

`: sql3-numeric?                 \ n1 -- flag`
True if column type *n1* is either `SQLITE_INTEGER` or `SQLITE_FLOAT`.

Pretty printing section

`40 Constant MAXWIDTH`
Maximun character width for a given column.

`32 Constant MAXCOLS`
Maximun number of columns to pretty-print.

`: th-colwidth                       \ n1 i --`
Set the *i*th column width to *n1*. Throw `SQL3ColumnIndex` if *i* out of bounds or `SQL3ColumnIndex` if *n1* is too wide.

`: colwidth                          \ n1 --`
Set all column widths to *n1*. Throw `SQL3ColumnIndex` if *n1* is too wide.

`: +colwidth                         \ n1 --`
Increment all column widths to *n1*. Throw `SQL3ColumnIndex` if the resulting width is too wide.

`SQLite3Callback: SQL-RESULT      \ client* nCols colValues* colNames* -- ior`
Print a query result set, like `|SQL.` but done as `|SQL> SQL-RESULT`. Printed formatting has deficiencies. Use `|SQL.` instead.