Preface and Information

Information

Version: 1.3.0
Author: Reko Tiira [ramirez]
E-Mail: reko@tiira.net
Date: 18th August 2009
IRC: ramirez @ irc.undernet.org [#mircscripting], ramirez @ irc.swiftirc.net [
#msl]

Welcome to using mIRC SQLite

mIRC SQLite is an implementation of the <u>SQLite</u> library for mIRC. It requires mIRC version 6.2 or higher.

It offers a flexible and easy-to-use API (Application Programming Interface) which is strongly inspired by the PHP's SQLite API.

This documentation has two purposes, first and foremost it works as a reference and secondly it has a lot of useful information on how to use the library. This includes explanation of features such as handling binary data with mIRC SQLite, registering and using user defined functions and aggregates, and how to handle errors.

This documentation is solely for documenting mIRC SQLite and assumes you are familiar with SQL already. It doesn't teach you SQL. You can find SQLite's documentation for the SQL language here: <u>http://www.sqlite.org/lang.html</u>. If you're familiar with other SQL databases such as MySQL, but have no prior experience with SQLite, feel reassured. SQLite isn't that much different at all. However I strongly recommend you to read this short document so you'll get an understanding of what SQL features SQLite doesn't support: <u>http://www.sqlite.org/omitted.html</u>.

This library has no external dependencies aside from the script file and the DLL (msqlite.mrc and msqlite.dll respectively).

If you find any bugs, or have feedback and/or suggestions, please get in touch with me on IRC, or e-mail me at the above address.

IMPORTANT. When you send me an e-mail make sure to include the word "mIRC" or "SQLite" (without quotes) in the subject, otherwise my mail filter won't catch it.

Features

SQLite Key Features

You might be wondering why I've decided to write a library for SQLite, and not for MySQL for example. The following list of features hopefully speaks for itself. I have made it brief purposefully, in case you want to read more about SQLite's features, read this: <u>http://www.sqlite.org/different.html</u>.

- **Serverless!** SQLite doesn't require a server to run as a seperate process/service.
- No configuration needed!

You just connect to a specified database. That's it.

• Fast!

SQLite is fast! To see user contributed speed comparisons, see: <u>http://www.sqlite.org/cvstrac/wiki?p=SpeedComparison</u>.

• Portable!

Because SQLite stores a database in a single file, it's very easy move databases around.

• Compact!

SQLite is extremely small in size, which makes it painless to download.

• Dynamic types!

SQLite implements so called manifest typing. You can insert any type of data to a table, regardless of the declared type.

This is especially good for simplicity's sake, as you don't even need to declare a data type for columns.

• Variable-length records!

Unlike many other SQL database engines, SQLite only stores as many bytes it needs.

This results into smaller and more efficient database files.

mSQLite Key Features

So what about mIRC SQLite then? What does it offer?

• Easy to use!

mSQLite provides a complete, easy to use and throughly documented API to interface with SQLite.

• Hash tables!

When fetching rows mSQLite will store the results efficiently in a hash table.

• Flexible error handling!

Every single mSQLite command updates the error variables, so you don't need to spend hours of time to debug an error.

• Regular expressions!

SQLite doesn't support regular expression matching by default, but mSQLite adds support for them. You can use the REGEXP keyword to do a regular expression matches.

• Binary data!

mSQLite extends the SQLite query syntax by implementing support for mIRC binary variables.

• User defined functions!

mSQLite allows users to register their own functions and aggregates to SQL with an ease.

• Garbage collecting!

In case you forget to free some resources, mSQLite will free them for you upon unloading!

Installing and Configuring mSQLite

Installing

Installing mSQLite is easy and quick.

After extracting the contents of the archive to a wanted destination, you should see these files:

msqlite.chm	Documentation	file.
msqlite.dll	DLL file.	
msqlite.ini	Configuration	file.
msqlite.mrc	Script file.	

If you do, the next step is to load the script in mIRC. Assuming you extracted the archive to mIRC folder, you can load mSQLite with:

/load -rs msqlite.mrc

If you extracted the archive to somewhere else than mIRC's root folder, you need to specify the path to the folder as well, for example:

/load -rs C:\scripts\msqlite\msqlite.mrc

If everything went correctly, mIRC should popup a message box asking whether you want to run the initialization script. You should choose Yes, although choosing No will not break the library either; It will just use the default configuration settings. That's it, you're done!

Configuring

mSQLite has a couple of configuration options in its configuration file (msqlite.ini). They are the following:

[MSQLITE] id An unique identifier for the library. busy_timeout Default busy timeout for new connections, in millis

The *id* option can be used to tell mSQLite to use a specific id for the library. By default id option is empty, which means that mSQLite finds a free unique id automatically, but in case you want force it to use a specific id, this option can be used.

The *busy_timeout* option instructs mSQLite to wait for at least specified amount of milliseconds when a query tries to operate on a locked database. If mSQLite obtains access to the database before busy timeout occurs, the operation is executed normally, otherwise an error is returned and **%sqlite_errno** is set to **\$SQLITE_BUSY**. Read more about error handling in <u>Handling Errors</u>. You can change busy timeout per-connection basis with <u>sqlite_busy_timeout</u> command.

[PRAGMAS]

pragma Pragma query to be executed by default when new data. The *pragma* directive is useful if you want to change default functionality of the library for all connections, because pragmas specified in the configuration file will be run everytime mSQLite opens a new connection to a database. You can have multiple pragma directives, and they must be under *[PRAGMAS]* section.

[EXTENSIONS]

extension An extension to be loaded automatically when new da The *extension* directive can be used to automatically load extensions for all new database connections. This is useful if you have general purpose extensions that you want to be able to use without explicitly loading the extension with sqlite_load_extension for every connection. The extensions are opened from the script directory of mSQLite unless an absolute path is specified. You can have multiple extension directives, and they must be under *[EXTENSIONS]* section.

Managing

Even though SQLite is very easy to learn and it's simple to do all sorts of queries, such as creating tables, it's always nice to have a visual managment tool that you can use to view your database structure, browse data and so on. There are a great deal of such tools out there, some of which are free, and some that are not. Here's a list of database managment tools for SQLite, with their price (if any): <u>http://www.sqlite.org/cvstrac/wiki?p=ManagementTools</u>.

You most likely don't want to try every single one of them to find out which are good, so in case you just want to get started quick you can take a look at the free, nice managment tool I use, <u>SQLite Spy</u>. SQLite Spy doesn't have a visual editor for creating tables, but it's by far the best visual browser for SQLite I've used. Another free tool that looks very promising, with features such as SQL code completion is <u>SQLite Administrator</u>.

The <u>SQLite Wiki</u> also has a great deal of other useful tools, like for converting databases between different database formats, and information on SQLite and how to use it more effectively. So if you're interested in reading more, check it out.

A Complete Example

The following example demonstrates few common features and functions of mSQLite. The example is commented throughly, so it should be easy to follow. Please read the <u>Handling Errors</u> and <u>Writing Queries</u> How To's for a more complete understanding and details.

```
Example
/*
** mIRC SQLite Demonstration Script
* *
        A greeting script
** _____
** Save as sqlite_example.mrc and do
** /load -rs sqlite_example.mrc
* *
** This script demonstrates some
** features of the SQLite.
* *
** Usage:
** /greet add nick greet
** /greet del [nick]
** /greet list [nick]
** /greet find greet
* *
** For del and list nick can be a wildcard.
** For find greet can be a wildcard.
*/
/*
** On start open connection and make sure the table exists
** You do not need to close the db connection explicitly
** mSQLite will close it when the mIRC exits
*/
on *:START:{
  set %greet_db $sqlite_open(greet.db)
  if (!%greet_db) {
   echo 4 -a Error: %sqlite_errstr
    return
  }
  sqlite_exec %greet_db CREATE TABLE IF NOT EXISTS greets (nick UNIQUE,
}
/*
** When someone joins a channel, check if he has
** a greet set and display the greet if one exists
*/
on *:JOIN:#:{
 ; Make sure we are connected to db
```

```
if (!%greet_db) { return }
 ; Check if the nick has a greet
  var %safe_nick = $sqlite_escape_string($nick)
  var %sql = SELECT greet FROM greets WHERE nick = ' $+ %safe_nick $+ '
  var %request = $sqlite_query(%greet_db, %sql)
  if (!%request) {
    echo 4 -a Error: %sqlite_errstr
    return
  }
 ; If the nick has a greet, display it
 if ($sqlite_num_rows(%request)) {
   var %greet = $sqlite_fetch_single(%request)
   msg $chan $nick $+ : %greet
  }
  ; Free the result
 sqlite_free %request
}
/*
** Greet alias
*/
alias greet {
 ; Make sure we are connected to db
 if (!%greet_db) {
    echo 4 -a Error: Not connected to greet.db
   return
  }
  ; OK! We're connected!
 var %cmd = $1
  if (%cmd != add && %cmd != del && %cmd != list && %cmd != find) {
   echo 4 -a Error: Unknown action.
    return
  }
 ; Do action
 if (%cmd == add) {
   ; Add greet
   var %nick = $2, %greet = $3-
   if (!%nick || !%greet) {
      echo 4 -a Error: Invalid arguments.
      return
   }
    ; Escape nick and greet and then execute query
   var %safe_nick = $sqlite_escape_string(%nick), %safe_greet = $sqlit
    sqlite_exec %greet_db REPLACE INTO greets (nick, greet) VALUES (' $
```

```
echo 3 -a Greet of %nick set to %greet
}
elseif (%cmd == del) {
  ; Delete greet
  var \%nick = $2
  if (!%nick) {
    ; Make sure you really want to delete everything
    var %del_all = $?!="You're about delete all greets in the databas
    if (!%del_all) { return }
    %nick = *
  }
  ; Replace wildcard * with % and escape nick and finally execute que
  var %safe_nick = $replace($sqlite_escape_string(%nick),*,%,?,_)
  sqlite_exec %greet_db DELETE FROM greets WHERE nick LIKE ' $+ %safe
  echo 3 -a $sqlite_changes(%db) greets deleted.
}
elseif (%cmd == list) {
  ; List greets
  var %nick = $2
  if (!%nick) { %nick = * }
  ; Replace wildcard * with % and escape nick
  var %safe_nick = $replace($sqlite_escape_string(%nick), *,%,?,_)
  ; Construct query and execute it
  var %sql = SELECT * FROM greets WHERE nick LIKE ' $+ %safe_nick $+
  var %request = $sqlite_query(%greet_db, %sql)
  if (!%request) {
    echo 4 -a Error: %sqlite_errstr
    return
  }
  ; List the greets
  echo -a -
  echo -a Total Greets Found: $sqlite_num_rows(%request)
  echo -a -
  ; For each row display the nick and greet
  while ($sqlite_fetch_row(%request, row)) {
    echo -a $hget(row, nick) - $hget(row, greet)
  }
  echo -a -
  ; Free the result
  sqlite_free %request
}
else {
  ; Find greets
  var %greet = $2-
  if (!%greet) {
```

```
echo 4 -a Error: Invalid arguments.
    return
  }
  ; We want to search for text even if it's in middle of the quote
  %greet = $+(*,%greet,*)
  ; Replace wildcard * with % and escape greet
  var %safe_greet = $replace($sqlite_escape_string(%greet), *, %, ?, _)
  ; Find text from greets
  var %sql = SELECT * FROM greets WHERE greet LIKE ' $+ %safe_greet $
  var %request = $sqlite_query(%greet_db, %sql)
  if (!%request) {
    echo 4 -a Error: %sqlite_errstr
    return
  }
  ; List the greets
  echo -a -
  echo -a Total Greets Found: $sqlite_num_rows(%request)
  echo -a -
  ; For each row display the nick and greet
  while ($sqlite_fetch_row(%request, row)) {
    echo -a $hget(row, nick) - $hget(row, greet)
  }
  echo -a -
  ; Free the result
  sqlite_free %request
}
```

}

Version History

1.3.0 - 18th August 2009

- Starting from this version mIRC SQLite will open databases and files from the directory specified by \$mircdir by default (earlier it used the directory where \$mircexe was in). Due to this change in mIRC installations where \$mircexe is not in \$mircdir, databases needs to be moved to \$mircdir if the scripts open databases by using relative paths. The reason for this change was that the DLL wasn't consistent with mIRC's built-in aliases and identifiers, because they used different directory than mIRC SQLite.
- Updated SQLite to version 3.6.17. Upgrading is recommended.
- Updated PCRE to version 7.9.
- Added <u>/sqlite_bind_null</u>.
- Added <u>\$sqlite_fetch_num</u>.
- Added <u>\$sqlite_fetch_assoc</u>.
- Added <u>\$sqlite_fetch_all</u>.
- Added <u>\$sqlite_safe_encode</u>.
- Added <u>\$sqlite_safe_decode</u>.
- Fixed <u>\$sqlite_fetch_bound</u> fetching one byte too little when fetching nonbinary data from an unbuffered query result into a binary variable.
- Fixed <u>\$sqlite_fetch_bound</u> not returning \$null when there's an error.
- Fixed <u>\$sqlite_fetch_bound</u> not returning 0 when there's no more rows to fetch.
- Fixed <u>\$sqlite_bind_field</u> not returning 1 on success.
- Fixed problems with passing binary data if there was a space in the temporary path used by Windows.
- Fixed "* /bread: invalid parameters" if trying to fetch binary data that is empty.
- Changed <u>\$sqlite_fetch_single</u> and <u>\$sqlite_fetch_field</u> return maximum string length from 900 bytes to 4096 bytes for mIRC versions 6.32+.

1.2.1 - 1st March 2009

- Updated SQLite to version 3.6.11. Upgrading is recommended.
- Fixed <u>\$sqlite_prepare</u> failing to prepare any statements. This critical bug was introduced in 1.2.0 (thanks to Sprak for reporting it).
- Changed SQLite to be compiled with SQLITE_ENABLE_UPDATE_DELETE_LIMIT flag, which allows user to use ORDER BY and LIMIT clause on UPDATE and DELETE statements. This was already changed in 1.2.0, but there was no note for it in the change

log.

1.2.0 - 22nd December 2008

- Due to the few major additions, changes and bug fixes in this version, this version will be called 1.2.0. Upgrading is strongly recommended.
- Added <u>\$sqlite_exec_file</u> for executing a file as SQL.
- Added a .file property to <u>\$sqlite_exec</u> for executing a file as SQL.
- Added a .file property to <u>\$sqlite_query</u> for executing a file as SQL.
- Added a .file property to <u>\$sqlite_unbuffered_query</u> for executing a file as SQL.
- Added a .file property to <u>\$sqlite_prepare</u> for executing a file as SQL.
- Added a feature to bind parameters for non-prepared statements using <u>\$sqlite_exec</u>, <u>\$sqlite_query</u> and <u>\$sqlite_unbuffered_query</u>. This will make writing cleaner queries much easier. See <u>Prepared Statements</u> page for example.
- Updated SQLite to version 3.6.7. This version adds new functionality and fixes a lot of important bugs from the previous SQLite version the library used, upgrading is strongly recommended!
- Updated PCRE to version 7.8.
- Fixed <u>\$sqlite_open</u> when opening a database from an existing file. It now creates triggers, views, indices and sequence values correctly.
- Fixed a bug with parameter binding for <u>/sqlite_exec</u> and <u>\$sqlite_unbuffered_query</u>.
- Fixed a critical bug with the REGEXP operator implementation (thanks to Sir-Loopy for reporting it).
- Fixed a bug with the autocommit feature.
- Changed <u>\$sqlite_write_to_file</u>. It now uses a custom written function which is more portable than the previous solution.

1.1.2 - 22nd April 2008

- Updated SQLite to version 3.5.8.
- Fixed a serious bug that caused <u>/sqlite_free</u> to stop working correctly. This bug was introduced in 1.1.0, upgrading is strongly recommended.

1.1.1 - 5th March 2008

- Added <u>/sqlite_help</u>.
- Added <u>/sqlite_autocommit</u>.
- Updated <u>\$sqlite_query</u>, <u>/sqlite_exec</u> and <u>\$sqlite_unbuffered_query</u> to support binding values to prepared statements.
- Updated SQLite to version 3.5.6.

- Updated PCRE to version 7.6.
- Fixed a serious bug that caused <u>/sqlite_rollback</u> to commit instead of rollback.

1.1.0 - 13th January 2008

- Due to the major additions, changes and bug fixes in this version, this version will be called 1.1.0. Upgrading is strongly recommended.
- Added support for prepared queries and binding parameters. For information about various binding styles that are supported by mSQLite, see <u>Prepared Statements</u>.
- Added <u>\$sqlite_unbuffered_query</u>.
- Added <u>\$sqlite_prepare</u>.
- Added <u>\$sqlite_bind_field</u> and its alias <u>\$sqlite_bind_column</u>.
- Added <u>\$sqlite_bind_param</u>.
- Added <u>\$sqlite_bind_value</u>.
- Added <u>\$sqlite_clear_bindings</u>.
- Added <u>\$sqlite_fetch_bound</u>.
- Added <u>\$sqlite_current_bound</u>.
- Added <u>\$sqlite_is_valid_statement</u>.
- Added <u>\$sqlite_begin</u>.
- Added <u>\$sqlite_commit</u>.
- Added <u>\$sqlite_rollback</u>.
- Added <u>\$sqlite_finalize</u>.
- Updated <u>\$sqlite_exec</u> and <u>\$sqlite_query</u> to be able to execute prepared statements.
- Updated <u>\$sqlite_fetch_row</u> to set **\$SQLITE_NOMOREROWS** as an error code when there are no more rows.
- Updated <u>\$sqlite_open</u> and <u>\$sqlite_open_memory</u> to also copy triggers and views when copying database from another file.
- Updated SQLite to version 3.5.4. There are a lot of important bug fixes inbetween this version and 3.4.1. Upgrading is strongly recommended.
- Updated PCRE to verion 7.4.
- Updated mSQLite to enable the FTS3 module in core by default.
- Fixed a long standing bug in <u>\$sqlite_query</u> not returning 0 when the query is a type of query that never returns rows (INSERT/UPDATE/DELETE).
- Fixed a bug where user defined functions that run nested queries removed temporary file created by TMPARG() prematurely.
- Removed FTS1 and FTS2 modules from the core to reduce size since they're obsolete. It's recommended to update your existing code to use

FTS3 instead. You can still use the older versions by compiling and loading the modules by yourself.

1.0.15 - 30th July 2007

• Fixed a critical bug that caused mIRC to crash upon loading a database when FTS2 module was enabled. Upgrading is strongly recommended.

1.0.14 - 30th July 2007

- Updated SQLite to version 3.4.1. This release fixes a bug in VACUUM that can lead to database corruption. The bug was introduced in version 3.3.14. Upgrading is recommended for all users.
- Updated PCRE to version 7.2.
- Updated <u>\$sqlite_create_aggregate</u> to have distinct properties for step and finalize alias. This allows users to create custom aggregates with only one mIRC alias by checking the \$prop identifier's value. See <u>User Defined</u> <u>Aggregates</u> for details and example.
- Fixed a lot of old information on documentation and added some new information on how-to pages for new users.

1.0.13 - 18th June 2007

- Updated SQLite to version 3.4.0. Fixes two seperate bugs that can cause a database corruption and adds explicit <u>limits</u> on the sizes and quantities of things SQLite will handle.
- Updated configuration file to automatically set temporary storage to memory instead of file. Before this was done by modifying the SQLite library source code, which meant that the corresponding edit had to be done for every version which is easy to forget. This ensures that temporary storage will be memory for every version unless you change it yourself.
- Updated the mIRC script to use an alias for the DLL path so people who wish to change where the DLL resides can do so easily by editing the corresponding alias.

1.0.12 - 27th April 2007

- Updated SQLite to version 3.3.17. Performance improvements added in 3.3.14 but mistakenly turned off in 3.3.15 have been reinstated. A bug has been fixed that prevented VACUUM from running if a NULL value was in a UNIQUE column. This version fixes a bug in the forwards-compatibility logic of SQLite that was causing a database to become unreadable when it should have been read-only.
- Updated PCRE to version 7.1.

1.0.11 - 15th April 2007

• Updated SQLite to version 3.3.15. An annoying bug introduced in 3.3.14 has been fixed. There are also many enhancements to the test suite.

1.0.10 - 8th April 2007

- Fixed a bug in custom functions argument encoder.
- Updated SQLite to version 3.3.14. This version focuses on performance improvements, you can see performance improvements up to 35% or more compared to the previois version. This version also adds support for <u>exclusive access mode</u>.
- Updated mSQLite to enable the <u>FTS2</u> (new version of the <u>Full-text Search</u>) module in the core by default.

1.0.9 - 15th February 2007

• Updated SQLite to version 3.3.13. This version fixes a subtle bug in the ORDER BY optimizer that can occur when using joins. There are also a few minor enhancements. Upgrading is recommended.

1.0.8 - 30th January 2007

- Fixed \$SQLITE_NOTMEMORYDB returning wrong value.
- Fixed <u>\$sqlite_fetch_field</u> outputting not intended debug information on certain errors.
- Updated the library to automatically find unique identifier for mSQLite if user doesn't specify one. You're no longer prompted to input an unique id on script load, but you can still force a certain id to be used by specifying it in msqlite.ini.
- Updated SQLite to version 3.3.12. This update fixes several bugs that were introduced in 3.3.10, upgrading is recommended.

1.0.7 - 14th January 2007

- Fixed some minor memory leaks in <u>\$sqlite_query</u> and <u>\$sqlite_exec</u> when queries failed.
- Updated SQLite to version 3.3.10. This update fixes several bugs that were introduced in 3.3.9, upgrading is recommended.

1.0.6 - 6th January 2007

- Updated SQLite to version 3.3.9. This update contains important changes and bug fixes, upgrading is recommended.
- Updated PCRE to version 7.0.

1.0.5 - 4th December 2006

- Added <u>\$sqlite_set_authorizer</u>.
- Fixed script failed to automatically unload when tried to load on older mIRC and script filename had spaces.
- Fixed various typos and misinformation in documentation.

1.0.4 - 3rd November 2006

- Fixed <u>\$sqlite_open</u> not supporting files or directories with spaces on them.
- Fixed mSQLite temporary tables storage using files instead of memory. This was misconfigured in version 1.0.3.
- Updated <u>\$sqlite_fetch_field</u> and <u>\$sqlite_result</u> to use case-insensitive matching when using field names.

1.0.3 - 11th October 2006

- Added <u>\$sqlite_result</u>.
- Updated SQLite to version 3.3.8, which adds support for full-text search. See <u>this article</u> for details.

1.0.2 - 5th September 2006

- Added <u>\$sqlite_open_memory</u>.
- Added <u>\$sqlite_write_to_file</u>.
- Added <u>\$sqlite_fetch_field</u>.
- Added <u>\$sqlite_is_valid_db</u>.
- Added <u>\$sqlite_is_valid_result</u>.
- Added <u>\$sqlite_is_memory_db</u>.
- Added <u>/sqlite_reload</u>.
- Added possibility of opening transient databases by using empty file name for <u>\$sqlite_open</u>.
- Added possibility of opening memory databases by using :memory: as file name for <u>\$sqlite_open</u>.
- Added new configuration options for loading extensions and running pragmas by default. See <u>Installing and Configuring mSQLite</u> for details.
- Fixed a bug opening msqlite.ini from mIRC dir, instead of the script dir.
- Fixed a bug in the ini parser, where it did not correctly remove enclosing quotes around value.
- Fixed <u>\$sqlite_rewind</u> not working.
- Fixed various typos and mistakes in documentation.
- Updated to store temporary tables on memory instead of temp files by default. This is much more efficient.

- Updated <u>\$sqlite_fetch_single</u> to return size of binary variable, instead of 1, when binvar is specified.
- Updated <u>\$sqlite_load_extension</u> to use native SQLite API instead of query for improved efficiency. You can still use *load_extension* in queries.

1.0.1 - 23rd August 2006

- Added <u>\$sqlite_field_metadata</u>.
- Added <u>\$sqlite_load_extension</u>.
- Fixed <u>\$sqlite_open</u> not setting the busy timeout to the default value specified in msqlite.ini.
- Updated SQLite to version 3.3.7.
- Updated PCRE to version 6.7.
- Updated to allow queries to load dynamic extensions; Starting from version 3.6.7 SQLite allows users to <u>dynamically load extensions</u>. *Note. This feature is still beta*.

1.0.0 - 6th August 2006

• Initial release.

Frequently Asked Questions

This FAQ also contains a number of technical questions and answers that might not interest people who don't develop mIRC libraries themselves. This is why I'll try to arrange the FAQ so that technical questions come after more general questions. If your question is not answered here or in the <u>Official SQLite FAQ</u>, feel free to contact me with your question, I also gladly answer technical questions.

IMPORTANT. When you send me e-mail make sure to include the word "mIRC" or "SQLite" (without quotes) in the subject, otherwise my mail filter won't catch it.

Q: I don't know SQL. Where can I learn it?

A: There a bunch of tutorials and documentation for SQL and SQLite on the Internet. Just use your favorite search engine and you'll know SQL in no time! The SQLite docs for SQL can be found here: <u>http://www.sqlite.org/lang.html</u>

Q: Should I use single or double quotes for string literals?

A: Single quotes. Double quotes are a dangerous thing in SQLite, because if the quoted string is an identifier, it will be treated as an identifier, otherwise as a string literal. This means that SQLite might interpret a string literal as an identifier when you mean to use it as a string literal. Not only that, but if you want to use double quotes to denote an identifier, and the identifier doesn't exist it will be automatically treated as a string literal. This means that SQLite might not report an error if you, for example, make a typo in your query. It's best to avoid double quotes completely.

Q: How do I tell SQLite that something is an identifier, such as a table or a column name?

A: You can usually leave it unquoted. If however the identifier is one of the reserved keywords that must be quoted to be used, use either square brackets or backticks. The list of keywords that must be quoted in order to be used as an identifier can be seen here: <u>http://www.sqlite.org/lang_keywords.html</u>.

Q: How can I make LIKE case sensitive? I tried '*a*' *LIKE BINARY* '*A*' but it doesn't work.

A: You can make LIKE case sensitive with the case_sensitive_like pragma. You can set it on with the following query: *PRAGMA case_sensitive_like=ON* Another option would be to use the *GLOB* function instead, which is always case sensitive. Please note that *GLOB* uses the Unix file globbing syntax for its wildcards, that is, it uses * instead of % and ? instead of _

Q: How can I make REGEXP case insensitive?

A: Use the feature to change matching options on fly. To make it use case insensitive matching, use *(?i)*, example: '*a*' *REGEXP* '*(?i)A*' You can reset the matching to act in a case sensitive manner by doing *(?-i)*

Q: Why do the row indexes start from 1 and not 0?

A: The reason is consistency. I wanted this library to be as consistent with mIRC as possible. And because in mIRC pretty much everything starts from 1 instead of 0 when it comes to indexing, I decided to do the same for mSQLite.

Q: Is unicode supported?

A: Yes. mSQLite uses UTF-8 encoding for a lot of things, such as for user defined function names or regular expressions. You can still insert text that isn't encoded in UTF-8 to SQLite databases. When using *REGEXP* to do a regular expression match, mIRC SQLite will first check the UTF-8 validity of the pattern and the subject. If they are valid, they are treated as unicode, otherwise they are not.

Q: I want to take advantage of unicode in regular expressions, how do I ensure that my text is valid UTF-8?

A: mIRC has a few useful identifiers for dealing with UTF-8. Namely you can use *\$utfencode* and *\$utfdecode* to encode and decode text respectively. *\$isutf* can be used to check for UTF-8 validity of text, it returns 0 if text isn't valid UTF-8, 1 if it is valid UTF-8 and only consists regular ASCII characters or 2 if it is valid UTF-8 and consists multibyte characters.

Q: If I open a file using relative path instead of absolute path, is the current directory mIRC directory or the script directory?

A: The mIRC directory. If you want to ensure the database is opened where the script is installed, and not in mIRC's root directory, use the \$scriptdir alias, that returns absolute path to the script directory.

Q: Since every database is stored in its own file, can't I use the same connection to query data from more than one database?

A: You can! Use the <u>ATTACH DATABASE</u> statement for this.

Q: How can I get a list of tables in a database?

A: Every SQLite database has a special table named sqlite_master that defines the database schema. You can use the following query to get a list of all tables in a database: *SELECT name FROM sqlite_master WHERE type='table'*

Q: Are nested queries allowed?

A: If you mean SQL subqueries, the answer is yes. If you mean a query in an user defined function, the answer is still yes. However you need to be careful in

this case, if your user defined function executes a query that calls the same user defined function, you might end up having an infinite loop. This can cause a crash.

Q: How does mSQLite find out whether the data passed to TMPRES is text or binary?

A: If the passed data contains NUL (0x0) character(s) or starts with SOH (0x1) it is considered as binary data, otherwise text.

Q: Your code is syntax highlighted, did you do all of that manually?

A: Hell no. All of the syntax highlighted code is generated automatically with a mIRC syntax highlighter script by Tye Shavik. The script is available here: <u>http://www.mirc.net/tye/mirc_script.phps</u>. Thanks a ton Tye!

Q: I can see that your mIRC code uses XML docs. Where can I get the tool to generate HTML docs from the XML and who made it?

A: The XML docs generator is a tool I created to make writing docs for this library less tedious. It's not available publicly as it's very unpolished, but I might release it some day.

Q: I appreciate your hard work, but there already is a SQLite wrapper DLL for mIRC. Didn't you know that?

A: As a matter of fact, I did. There are many reasons why I decided to make my own SQLite library for mIRC, even though knowing one existed already. First and foremost, the SQLiteDll wrapper is terribly outdated, mSQLite brings you the features of the latest SQLite. Second of all I wanted to make the library more versatile and feature rich. Here are some features that mSQLite adds to those of SQLiteDll: A more user friendly API which exposes more functionality to the user, SQL regular expressions with the REGEXP keyword, handling binary data, user defined functions, fetching rows to hash tables instead of using an unsafe and limited seperator mechanism and more.

Q: Where is sqlite_unbuffered query?

A: As of version 1.1.0 unbuffered queries are supported.

Q: Why isn't there a way to open a persistent connection in mSQLite, as you can do with sqlite_popen in PHP?

A: In mSQLite it's not necessary. In PHP the advantage of using a persistent connection means that when a connection is established to an already open database, the schema data needs not to be re-read. mSQLite uses a shared cache for databases and schema data for connections to the same database. This means that mSQLite implements persistent connections implicitly, and even improves the PHP's persistent connection mechanism by implementing reference counting;

In other words, a call to sqlite_close on an open persistent connection doesn't close the connection to the database if there are other active connections.

Q: Why are you required to have an unique id for every instance of mIRC and mSQLite?

A: The reason for this is because mSQLite uses mIRC's SendMessage API to intercommunicate with mIRC. mIRC implements intercommunication with SendMessage by using named mapped files. If every instance of mIRC used the same mapped file, there could be problems with two instances of mIRC trying to access the same mapped file at same time. This would result into undefined behavior. To prevent such cases, one would need to implement synchronization to the mapped file, but this would have its own share of problems, one being inefficiency. While there are other ways to intercommunicate with mIRC, such as using DDE, it is much more inefficient and has its own share of limitations of use. By using an uniquely named mapped file, mSQLite can have an exclulsive access to the mapped file, which is the safest and most efficient way to intercommunicate with mIRC. This is also the reason why mSQLite wasn't released until now. I was trying to find a decent way to make mSQLite safe even when multiple instances of mIRC are running or other scripts require access to the mapped file. Then mIRC 6.2 was released and with some modifications I accomplised the goal in the most efficient way I could've hoped.

Q: I understand why you're required to have an unique id now. But why don't you just generate one instead of forcing the user to give one?

A: As of version 1.0.8 this is exactly what mSQLite does! You can still give the id manually if you wish though, by editing the configuration file.

Q: What implementation of regular expressions does mSQLite use? A: mSQLite uses <u>PCRE</u> (Perl Compatible Regular Expressions) for regular expressions implementation.

Q: Can I have the source code?

A: I have no intention of releasing the source code for as long as I personally work on the project.

Handling Errors

mIRC SQLite has a simple and effective way to do error handling. Unlike in many APIs, there is no function to query the database for the last error. Instead every mSQLite function sets two special variables that can be examined to determine the cause of an error:

%sqlite_errno Holds the error code of the latest operation. %sqlite_errstr Holds the error string of the latest operation. There is also one special function that can be used to get a string representation of an error code, <u>\$sqlite error string</u>.

The returned string isn't necessarily the same as the *%sqlite_errstr*. Sometimes *%sqlite_errstr* contains a specific error message when an error is returned. For example, whenever *%sqlite_errno* is set to *\$SQLITE_INVALIDARG*, *%sqlite_errstr* will contain a specific error message of which argument is invalid and why. Similiarly when opening a database or executing a query fails, *%sqlite_errstr* will contain the specific error message, while *%sqlite_errno* contains the type of the error.

There are two ways to determine if there was an error in a mSQLite command or identifier. The first way is to check the return value of the identifier, which is always **\$null** in case of an error. In case *\$null* is returned, you can then use the *%sqlite_errno* and *%sqlite_errstr* to determine the exact cause of the error. The other way to check for an error is to examine the variables after the call to the function.

NOTE. In case you call mSQLite functions as commands (eg. /sqlite_* instead of **\$sqlite_***), you can also use the **\$result** identifier of mIRC to see what was returned.

Error codes

Here is the list of error codes that *%sqlite_errno* can consists:

	i i i i i i i i i i i i i i i i i i i	
Identfier	Description	Cod
\$SQLITE_OK	not an error	1
\$SQLITE_ERROR	SQL logic error or missing database	
<pre>\$SQLITE_INTERNAL</pre>	an internal logic error in SQLite	
<pre>\$SQLITE_PERM</pre>	access permission denied	1
\$SQLITE_ABORT	callback requested query abort	
\$SQLITE_BUSY	database is locked	ļ
\$SQLITE_LOCKED	database table is locked	1
<pre>\$SQLITE_NOMEM</pre>	out of memory	
\$SQLITE_READONLY	attempt to write a readonly database	ł
\$SQLITE_INTERRUPT	interrupted	!

\$SQLITE_I0ERR	disk I/O error	1
\$SQLITE_CORRUPT	database disk image is malformed	1:
\$SQLITE_FULL	database or disk is full	1
<pre>\$SQLITE_CANTOPEN</pre>	unable to open database file	1
\$SQLITE_PROTOCOL	database locking protocol failure	1!
\$SQLITE_SCHEMA	database schema has changed	1
\$SQLITE_TOOBIG	too much data for a row	1:
<pre>\$SQLITE_CONSTRAINT</pre>	constraint failed	1!
<pre>\$SQLITE_MISMATCH</pre>	datatype mismatch	2
\$SQLITE_MISUSE	library routine called out of sequence	2:
\$SQLITE_NOLFS	kernel lacks large file support	2:
\$SQLITE_AUTH	authorization denied	2
<pre>\$SQLITE_FORMAT</pre>	auxiliary database format error	2
<pre>\$SQLITE_RANGE</pre>	bind or column index out of range	2
\$SQLITE_NOTADB	file is encrypted or is not a database	2
\$SQLITE_INVALIDARG	invalid argument	20
<pre>\$SQLITE_NOMOREROWS</pre>	no more rows available	20
\$SQLITE_NOTMEMORYDB	not a memory database	20

You should always use the identifiers instead of code values when comparing error codes. This is because the error codes are a possible subject to change in future. By using the identifier you'll ensure compatibility with future versions.

Writing Queries

Because of the lack of quote delimited strings in mIRC you need to write your queries with care to ensure there aren't any problems. Here I will include few guidelines for writing queries in a clean, safely manner.

• Pre-assign queries.

When you're executing a query with an identifier such as <u>\$sqlite_query</u> you should avoid writing the queries inside the identifier. The reason for this is because characters like (,) and , won't work properly, for example mIRC always interprets comma as argument seperator. To use comma properly inside the identifier, you'd need to escape it with *\$chr(44)*. This produces messy and sometimes even unreadable queries.

There's a simple solution for this, just pre-assign the query to a variable, and then pass the variable as the parameter to the identifier. You can either preassign with */set* or */var*, sometimes there might be problems with */var* as it allows assignment of multiple local variables at same time, thus if mIRC sees a comma followed by a variable, it thinks you're declaring another variable.

The following code should clarify this:

Example 1

```
; BAD! mIRC misinterprets the comma between col1 and col2
; It is part of the query, but mIRC thinks it seperates args for $:
var %request = $sqlite_query(%db, SELECT col1, col2 FROM table)
```

```
; GOOD! No misinterpration this time
var %sql = SELECT col1, col2 FROM table
var %request = $sqlite_query(%db, %sql)
```

Example 2

```
; BAD! mIRC misinterprets the comma followed by %col2
; It thinks you're declaring a list of local variables
var %sql = INSERT INTO table (col1, col2) VALUES ( %var1 , %var2 )
; GOOD! Unlike /var, /set only allows one variable at time, so the
set %sql INSERT INTO table (col1, col2) VALUES ( %var1 , %var2 )
```

Keep in mind that /*set* sets a global variable instead of local variable. This should be fine if you use the same variable name (eg. *%sql*) everywhere, so your global variable list won't be polluted. You can also use /*set -l* to set a single local variable, without the danger of mIRC misinterpreting commas (thanks for the tip Chad!)

NOTE. When queries are executed as a command with <u>/sqlite_exec</u>, it isn't required to pre-assign the query, as none of the above problems are present.

• Use spaces and pre-quoting.

Unlike most languages mIRC has no limitations for what characters, except for a space, a variable name can support. This means that whenever you write queries you need to make sure that variable names are detached from characters that are part of the SQL query, not the variable name. For numeric types this usually is very easy, see example:

```
Example 1
```

```
; BAD! mIRC doesn't treat %var1 as a variable because the word does
; and mIRC interprets %var2 so that the closing paranthesis is para
set %sql INSERT INTO table (col1, col2) VALUES (%var1, %var2)
; GOOD! Variable names are now seperated by spaces
set %sql INSERT INTO table (col1, col2) VALUES ( %var1 , %var2 )
```

However, for string data this isn't quite as simple. The reason is because if you do '%var' mIRC doesn't recognize the variable because the word doesn't start with %, but in the other hand if you do ' %var ' the spaces will be includes in the string. You either need to connect the quotes to the variable with \$+ or prequote the data with <u>\$sqlite_qt</u>, as following:

Example 2

```
; BAD! mIRC doesn't treat %str1 as a variable because the word does
; For col2 the spaces will be included in the string, which isn't v
set %sql INSERT INTO table (col1, col2) VALUES ('%str1', ' %str2 ')
; GOOD! Now col1 and col2 both are quoted correctly
set %sql INSERT INTO table (col1, col2) VALUES (' $+ %str1 $+ ', $-
; GOOD! Prequoting the strings will fix the problem too
var %str1 = $sqlite_qt(%str1), %str2 = $sqlite_qt(%str2)
set %sql INSERT INTO table (col1, col2) VALUES ( %str1 , %str2 )
```

• Escape unsafe data.

Whenever dealing with string data that might be unsafe it should be escaped properly before using it in a query. Unsafe string is a string that might contain a string delimiter (eg. a single quote) in it. The reason why it's unsafe is because when a single quote is seen in a wrong place, SQLite will misinterpret where the data ends, and thus start executing the query from wrong offset. This is especially dangerous with user input, as it allows **SQL Injection**. To prevent this scenario, always use <u>\$sqlite_escape_string</u> on a potentially unsafe string.

Example of escaping user input proeprly:

```
Code
; BAD! Allows the user to input unescaped single quote, will break
; Example of SQL injection: !login ' OR user = 'admin
; Will generate: SELECT * FROM users WHERE user = 'nick' AND pass =
on *:TEXT:!login *:?:{
  var %user = $sqlite_qt($nick), %pass = $sqlite_qt($2-)
  set %sql SELECT * FROM users WHERE user = %user AND pass = %pass
  ; ...
}
; GOOD! By escaping the password, SQL injection is no longer possil
; Will generate: SELECT * FROM users WHERE user = 'nick' AND pass =
; In SQLite single quote is escaped by doubling it with another sin
on *:TEXT:!login *:?:{
  var %user = $sqlite_qt($nick), %pass = $sqlite_qt($sqlite_escape_
  set %sql SELECT * FROM users WHERE user = %user AND pass = %pass
  ; ...
}
```

• Write long queries on multiple lines.

mIRC allows you to span a single command on multiple lines with the *\$*& identifier. This can become very handy when writing long queries with tons of fields:

```
Code
```

```
; BAD! Too much text stuffed on single line
sqlite_exec %db INSERT INTO table (one, two, three, four, five) VAI
; GOOD! Looks nicer on multiple lines
sqlite_exec %db $&
  INSERT INTO table ( $&
   one, two, three, four, five $&
   ) VALUES ( $&
   %one , %two , %three , %four , %five $&
   )
```

• Escape special characters when necessary.

Even though the guidelines explained in previous sections make writing

queries cleaner and easier, there are times when some characters needs to be *escaped* in order for mIRC not to parse them. Here's a common example of such case:

```
Code

; BAD! mIRC interprets the closing %' as a variable, instead of a %

var %sql = SELECT * FROM games WHERE title LIKE '% $+ %title $+ %'

; GOOD! This will make mIRC interpret % as a percentage character,

var %sql = SELECT * FROM games WHERE title LIKE '% $+ %title $+ % $
```

Another way around this situation is to pre-format the LIKE clause:

```
Code
; GOOD! mIRC won't misinterpret percentage characters
; Note that we also pre-quote the data as explained before
var %like = $sqlite_qt(% $+ %title $+ %)
var %sql = SELECT * FROM games WHERE title LIKE %like
```

Escape Codes

The following table shows the special cases that needs to be escaped and the required code to escape them. Some characters have more than one way to escape them, they are all listed below each other:

```
Chr
       Escape
       $chr(44)
1
%var
       % $+ var
       $chr(37) $+ var
$id
       $!id
       $ $+ id
       $chr(36) $+ id
Γ
       $chr(91)
]
       $chr(93)
{
       $chr(123)
}
       $chr(125)
```

The characters [] { and } only needs to be escaped when seperated by spaces.

Handling Binary Data

One of the nicest features of mIRC SQLite is its ability to deal with binary data painlessly by interacting with mIRC binary variables.

Writing binary data to a database

mIRC SQLite extends the SQLite query language syntax by adding support for mIRC-like binary variables. This means that you can actually use **&bvar**; syntax inside queries! mIRC SQLite will then correctly translate the binary var to actual binary data and use it in queries. For example:

```
Code
```

```
sqlite_exec %db INSERT INTO binary_data (key, bytes) VALUES ('backup',
sqlite_exec %db UPDATE binary_data SET bytes = &bvar; WHERE key = 'back
```

That's it! Simple and effective.

Reading binary data from a database

To read binary data from a database you first need to query it:

```
Code
```

```
var %request = $sqlite_query(%db, SELECT bytes FROM binary_data WHERE k
```

You have then two ways to fetch it:

1. sqlite_fetch_row

When you fetch a result with binary data <u>\$sqlite_fetch_row</u> will create a binary variable on the specified hash table for the binary field.

```
Code
; When dealing with big chunks of binary data you should always spe
; If you don't mSQLite needs to take a copy of the same binary data
if ($sqlite_fetch_row(%request, row, $SQLITE_ASSOC)) {
    ; Read the binary data to a binary variable, returns the number of
    var %size = $hget(row, bytes, &bvar;)
}
```

2. sqlite_fetch_single

Fetching binary data with <u>\$sqlite_fetch_single</u> is a bit more simple, since it only fetches the first field. To fetch data to a binary variable, just specify the name of the binary variable as second argument. This changes the functionality of *sqlite_fetch_single* to return the size of the binary variable on success, instead of the column data.

Code

```
; Returns size of bvar on success
var %size = $sqlite_fetch_single(%request, &bvar;)
if (%size != $null) {
   ; We now have the binary data in &bvar;
}
```

That's all for reading binary data into binary variables. Don't forget to free the result with <u>sqlite_free</u> as big chunks of binary data can take quite a bit of memory.

Binary Data Example

The following example demonstrates how to work with binary data in mSQLite by copying a file. Please note that this example is purely for demonstrating purposes and shouldn't be used to copy files in real scripts, as there are much more simple and efficient ways of doing so. The code is throughly commented so it should be easy to follow.

Example

```
/*
** mIRC SQLite Demonstration Script
* *
            Binary data handling
* *
** Save as sqlite_binary.mrc and do
** /load -rs sqlite_binary.mrc
* *
** This script demonstrates how to
** use binary variables with mSQLite.
* *
** Usage:
* *
   /copyfile source destination
* *
** Try with:
* *
   /copyfile mirc.exe mirc.exe.backup
* *
** This simply copies the mIRC.exe file.
** Unpractical as there are better
** ways to copy a file, but it does
** so by writing the binary data to
** a SQLite database first and then
** querying the data from the database.
* *
** This example is only for demonstration
** purposes, so support for filenames that
** have spaces in their path is not added.
*/
alias copyfile {
```

```
var %src = $1, %dst = $2
if (!%src || !%dst) {
 echo 4 -a Error: Too few arguments.
  return
}
; Make sure source file exists
if (!$isfile(%src)) {
 echo 4 -a Error: File $qt(%src) doesn't exist.
  return
}
; Get the size of the source file and dest dir
var %size = $file(%src).size, %dir = $nofile(%dst)
if (!%dir) { %dir = $mircdir }
; Make sure it exists
if (!$isdir(%dir)) {
  echo 4 -a Error: Directory $qt(%dir) doesn't exist.
 return
}
; Make sure source and dest files aren't the same file
if (%src == %dst) {
 echo 4 -a Error: You can't copy the file to the same file.
 return
}
; All fine, open files database and create table for files, if one do
var %db = $sqlite_open(files.db)
sqlite_exec %db CREATE TABLE IF NOT EXISTS files (file, size)
; Start copy
echo 3 -a Copying...
; Read the contents of the source file to a binary variable and write
bread %src 0 %size &src;_bytes
sqlite_exec %db INSERT INTO files (file, size) VALUES ( &src;_bytes ,
; Make sure there wasn't an error
if (%sqlite_errno == $SQLITE_OK) {
 ; No errors
 echo 3 -a File written to database.
  ; Get the ID of the file
 var %file_id = $sqlite_last_insert_rowid(%db)
  ; Now let's query the db for the file
 var %sql = SELECT file, size FROM files WHERE ROWID = %file_id
 var %request = $sqlite_query(%db, %sql)
```

```
if (%request) {
      ; No errors
      echo 3 -a File read from database.
      ; Fetch the results
      if ($sqlite_fetch_row(%request, row)) {
       ; Get the bytes to a new binary variable, and make sure the siz
       %size = $hget(row, file, &dst;_bytes)
        if (%size == $hget(row, size)) {
          ; All good! Write the new file. :)
          bwrite %dst 0 %size &dst;_bytes
         echo 3 -a Done!
        }
        else {
         echo 4 -a Error: Size mismatch
        }
      }
      else {
        echo 4 -a Error: %sqlite_errstr
      }
      ; Free the query result
     sqlite_free %request
    }
   else {
     echo 4 -a Error: %sqlite_errstr
   }
 }
 else {
   echo 4 -a Error: %sqlite_errstr
 }
 ; Close db
 sqlite_close %db
}
```

Prepared Statements

As of version 1.1.0 mSQLite supports prepared statements. Prepared statements, in simple terms, are a way to prepare a query before actually executing it, you can think of it as a compiled SQL template of a sort. There are two major benefits of using prepared statements:

- In case you want to execute the same query, only with different parameters, more than one time the query only needs to be prepared (or compiled) once. This can offer a performance boost for complex queries.
- They're safe from SQL injections if parameter binding is used. In other words, when parameters are bound with the binding API that mSQLite supports, no data needs to be quoted, SQLite will handle it for you.

It's important to remember that prepared statements aren't always necessarily faster than unprepared statements, even when you execute a large set of same query. The complexity of the query is the major factor.

Getting started

To prepare a query you need to use <u>\$sqlite_prepare</u>. It works very much like <u>\$sqlite_query</u>, except that instead of returning a result set produced by the query, it returns a prepared statement.

Example of preparing a statement and then executing it:

```
Code
```

```
; Assumes that %db is already open, and table contacts exists in the da
var %sql = INSERT INTO contacts (name, email) VALUES ('Joe', 'joe@gmail
var %stmt = $sqlite_prepare(%db, %sql)
sqlite_exec %stmt
sqlite_free %stmt
```

Of course using prepared statements like this isn't useful. Their usefulness comes from being able to bind parameters to the SQL query. Binding parameters is a way to substitute certain parts of the query, with values specified with the binding API. Here's a the previous example but with more meaningful query:

```
Code
; Assumes that %db is already open, and table contacts exists in the da
var %sql = INSERT INTO contacts (name, email) VALUES (?, ?)
var %stmt = $sqlite_prepare(%db, %sql)
sqlite_bind_value %stmt 1 Joe
sqlite_bind_value %stmt 2 joe@gmail.com
sqlite_exec %stmt
sqlite_free %stmt
```

The values specified with question marks in the query are later substituted with *Joe* and *joe@gmail.com* respectively.

Binding parameters

This section is to show what different types of binding parameters mSQLite support. Each type of binding falls into one of the two categories:

• Automatic input binding

Even before version 1.1.0 mSQLite supported this type in form of binary variables. In 1.1.0 you can also bind regular variables with similiar syntax, using @ to prefix the variable name. Here's an example that shows how to bind a regular variable and a binary variable:

```
Code

set %image_name mIRC Logo

bread mIRC.png 0 $file(mIRC.png).size J_bytes

var %sql = INSERT INTO images (name, bytes) VALUES (@image_name, J

sqlite_exec %db %sql
```

Note that variables bound with automatic binding must be global variables and can only contain letters, numbers and underscores.

• Manual input binding

Manual input binding always requires a prepared statement. There are three ways to bind parameters: <u>sqlite_bind_param</u>, <u>sqlite_bind_value</u> and binding them as extra arguments for <u>sqlite_query</u>, <u>sqlite_exec</u> and

<u>sqlite unbuffered query</u>. The first way is similiar to automatic binding of variables, with the exception being that you can use more complex naming for the variables and you can specify the type of the variable if needed. Example:

Code

```
; Contact info
set %contact.name Joe
set %contact.tel 123456789
; Insert contact
var %sql = INSERT INTO contacts (name, tel) VALUES (:name, :tel)
var %stmt = $sqlite_prepare(%db, %sql)
sqlite_bind_param %stmt :name contact.name
sqlite_bind_param %stmt :tel contact.tel $SQLITE_TEXT
sqlite_exec %stmt
sqlite_free %stmt
```

With manual binding of variables, you can use variable names such as

%contact.name, that contain special characters like period. With automatic binding you can't do this. Another demonstrated feature that isn't possible with automatic binding is specifying the type for the bound variable. In this particular case we're telling mSQLite that the telephone number should always be considered a text. If we didn't do this, mSQLite would consider it as a number since it contains only numbers.

Both types of binding, <u>sqlite_bind_param</u> and <u>sqlite_bind_value</u>, support both named parameters and numeric parameters. Numeric parameters are specified with a question mark in the query, named parameters are prefixed with a colon. Here's a couple of examples to demonstrate them:

Code

```
; Numeric parameters
var %sql = INSERT INTO contacts (name, tel) VALUES (?, ?)
; Named parameters
var %sql = INSERT INTO contacts (name, tel) VALUES (:name, :tel)
; Numeric and named parameters
var %sql = INSERT INTO contacts (name, tel) VALUES (:name, ?2)
```

The third way of binding parameters to prepared statements is most likely the most intuitive one if you only need numeric parameters and don't want to re-use global variables as bound parameters nor call sqlite_bind_value a bunch of times each time you re-execute the statement. An example:

```
Code
; Insert contact
var %sql = INSERT INTO contacts (name, tel) VALUES (?, ?)
var %stmt = $sqlite_prepare(%db, %sql)
noop $sqlite_exec(%stmt, Joe Gibson, 123456789)
sqlite_free %stmt
```

Note that we use */noop* here because the name contains a space. By using *noop* and calling the *sqlite_exec* as an identifier instead of as a command, we can make mIRC tokenize the parameters correctly.

As of version 1.2.0 you can bind parameters to non-prepared

statements using the same intuitive syntax as above. The only limitation is that you must call the query functions using the identifier format (i.e. \$sqlite_query instead of /sqlite_query):

```
Code
; Insert contact
var %sql = INSERT INTO contacts (name, tel) VALUES (?, ?)
noop $sqlite_exec(%db, %sql, Joe Gibson, 123456789)
; Select all contacts with name Joe Gibson
var %sql = SELECT * FROM contacts WHERE name = ?
var %result = $sqlite_query(%db, %sql, Joe Gibson)
```

This is recommended way of inserting user data in queries, as it's the most clean and the most secure way to do it, and prevents the chance of SQL injections completely. Again, you can use */noop* if you just want to execute a query, but don't care for the return value.

All unbound parameters will be substituted with *NULL* when a query is executed.

• Output binding

The last type of binding is output binding. With output binding, you can fetch data from a result set directly into variables using <u>sqlite_fetch_bound</u> or <u>sqlite_current_bound</u>. To bind an output variable you use <u>sqlite_bind_field</u> or its alias <u>sqlite_bind_column</u>. You can bind column names using its numeric index, or its name. This is very straightforward, so here's an example to demonstrate the usage:

```
Code
var %sql = SELECT name, email, tel FROM contacts
var %result = $sqlite_query(%db, %sql)
; Bind using column index
sqlite_bind_field %result 1 contact.name
sqlite_bind_field %result 3 contact.tel
; Bind using column name
sqlite_bind_field %result email contact.email
; Fetch the contacts
while ($sqlite_fetch_bound(%result)) {
    echo -a Name: %contact.name - Email: %contact.email - Tel: %contact.email
}
```

You can also bind columns to binary variables, by prefixing the variable name with an ampersand as usual with binary variables. If you bind column with a binary data to a regular variable, the binary data will be converted into text and assigned to the variable. Similiarly if you bind a non-binary data to a binary variable, the binary variable will contain ASCII representation of the data.

Important guidelines

There are a few things you need to be aware when binding parameters. Here's a list of some things you should know.

• Do not escape strings yourself

SQLite will do it for you. In non-prepared statements, you generally have to use <u>\$sqlite_escape_string</u> to escape special characters in strings to prevent SQL injection. With prepared statements there's no need for this.

• You can not bind values inside text

Parameter binding has no effect inside strings. For example you can't do this:

```
Code
var %sql = SELECT * FROM contacts WHERE name LIKE '?@%'
var %stmt = $sqlite_prepare(%db, %sql)
sqlite_bind_value %stmt 1 joe
```

Instead, you should be doing this:

```
Code
var %sql = SELECT * FROM contacts WHERE name LIKE ?
var %stmt = $sqlite_prepare(%db, %sql)
sqlite_bind_value %stmt 1 joe@%
```

• Be extra cautious with variable names when using sqlite_bind_field

Because mSQLite has no access to local variables of aliases that mSQLite is being used from, it must bind variables bound with <u>sqlite_bind_field</u> as global variables. Because of this, you must be very careful with the variable names you use in order not to overwrite variables from other scripts. It's a good idea to prefix your variables in order to avoid name conflicts. Example:

Code

```
var %sql = SELECT name, email FROM contacts
var %result = $sqlite_query(%db, %sql)
sqlite_bind_field %result name contacts.name ; %contacts.name
sqlite_bind_field %result email contacts.email ; %contacts.email
```

• Use \$sqlite_bind_value instead of /sqlite_bind_value when the value might contain spaces

You must use the identifier form of sqlite_bind_value when the data that is

being bound might contain spaces. The reason is that the function contains another parameter after the value to be bound, and mIRC doesn't know how to deal with arguments containing more than one word when used as command. An example:

```
Code
; Don't do this!
sqlite_bind_value %stmt 1 Joe Johnson
; Do this!
noop $sqlite_bind_value(%stmt, 1, Joe Johnson)
```
User Defined Functions

mIRC SQLite allows the users to register two kinds of functions, regular functions and aggregates. This section explains the details of both, and shows how to implement them.

NOTE. User defined functions are not limited to custom functions. You can override the core functions and aggregates by registering new functions with their names.

You can use <u>sqlite_create_function</u> to register a mIRC alias as a SQL function. Please look at the reference to see an explanation of the parameters. Example of an user defined function:

```
Code
; Returns a specified amount of characters from left side of a md5 gene
alias md5_left {
  var %str = $1, %len = $2
  return $left($md5(%str), %len)
}
```

You would then register it with mSQLite:

```
Code
sqlite_create_function %db md5_left md5_left 2
```

After you have registered it with mSQLite you can use it in queries:

```
Code
; Outputs 5ac749fb
var %sql = SELECT md5_left('some string', 8)
var %request = $sqlite_query(%db, %sql)
if (%request) {
    echo -a $sqlite_fetch_single(%request)
}
```

But what if you wanted to make the length parameter optional? There are two ways for this.

```
Code
; Implementing optional arguments by registering the
; function twice with different amount of arguments
alias md5_left {
 ; Default length is 8
 var %str = $1, %len = $iif($0 == 1, 8, $2)
 return $left($md5(%str), %len)
}
```

```
; Then register it twice with different amount of args
sqlite_create_function %db md5_left md5_left 1
sqlite_create_function %db md5_left md5_left 2
```

Or you can check the number of arguments yourself and signal an error with <u>sqlite_signal_error</u>:

Code
; Implementing optional arguments by registering
; the function with any amount of arguments
alias md5_left {
 ; Make sure there are only 1 or 2 arguments, otherwise signal an erro
 if (\$0 !isnum 1-2) {
 return \$sqlite_signal_error(wrong number of arguments to function m
 }
 var %str = \$1, %len = \$iif(\$0 == 1, 8, \$2)
 return \$left(\$md5(%str), %len)
}
; Then register it with any number of arguments
sqlite_create_function %db md5_left md5_left -1

You can also take advantage of mIRC's built-in identifiers when registering user defined functions. For example you could implement base_convert as following:

Code

```
; convert_base(n, from, to) calls $base(n, from, to)
sqlite_create_function %db base_convert base 3
```

Finally, here's an example of using custom property in an user defined function:

```
Code
; sin_r calls $sin(alpha) using radians
sqlite_create sin_r sin 1
; sin_d calls $sin(alpha).deg using degrees
sqlite_create sin_d sin 1 deg
```

Function Example

This a complete example of the above *md5_left* user defined function. The example also adds more error checking in the user defined function itself.

Example

```
/*
** mIRC SQLite Demonstration Script
* *
               MD5_LEFT Function
** _____
** Save as sqlite_function.mrc and do
** /load -rs sqlite_function.mrc
* *
** This script demonstrates the user
** defined functions of the SQLite.
* *
** Usage:
** /pass_gen hash [ len ]
** Try with:
** /pass_gen
** /pass_gen s9j331almva
** /pass_gen +sfa9ufa?sa 8
** /pass_gen fafsafa35os -1
* *
** The /pass_gen will generate a simple
** password with md5. The len argument
** is optional, and if omitted 8 is assumed.
*/
alias pass_gen {
  var %db = $sqlite_open(empty.db)
  if (!%db) {
    echo 4 -a Error: %sqlite_errstr
    return
  }
  var %hash = $sqlite_escape_string(%hash), %len = $2
  ; Register the function with 1 or 2 args
  sqlite_create_function %db md5_left md5_left 1
  sqlite_create_function %db md5_left md5_left 2
  ; Do query
  if (%len !isnum) {
    set %sql SELECT md5_left(' $+ %hash $+ ')
  }
  else {
    set %sql SELECT md5_left(' $+ %hash $+ ', %len )
  }
  var %request = $sqlite_query(%db, %sql)
  if (%request) {
    var %pass = $sqlite_fetch_single(%request)
    echo -a Generated password: %pass
    sqlite_free %request
```

```
}
  else {
   echo 4 -a Error: %sqlite_errstr
  }
  sqlite_close %db
}
; User defined function
alias md5_left {
  ; Make sure that has isn't NULL
  var %str = $1
  if (%str == $null) {
   var %error = empty hash in md5_left()
   return $sqlite_signal_error(%error)
  }
  ; See if len was specified and that it's valid
  var %len = 8
  if ($0 == 2) {
    % len = $2
    if (%len !isnum 1-32) {
      var %error = invalid pass length in md5_left()
      return $sqlite_signal_error(%error)
    }
  }
 ; Return the generated pass
  return $left($md5(%str), %len)
}
```

User Defined Aggregates

You can use <u>sqlite_create_aggregate</u> to register a mIRC alias as a SQL aggregate. Please look at the reference to see an explanation of the parameters. User defined aggregates are very similiar to regular functions, so only the differences will be explained here, see <u>User Defined Functions</u> for more information. The key difference is that you need to actually register two mIRC aliases to create an aggregate. SQL aggregates are used to, as the name implies, aggregate data from rows, thus the need of two different aliases: One that keeps track of the aggregating, called the step function, and one that returns the final result of aggregation at the end, called finalizer function.

mSQLite passes an *aggregation context* to the step alias. The step alias can use this context to accumulate the result, and at the end of the alias it should return the new aggregation context.

Here's an example step alias for keeping track of max length of a string:

```
Code
alias max_len_step {
  var %context = $1, %len = $len($2)
  if (%context !isnum || %len > %context) {
    %context = %len
  }
  return %context
}
```

The *%context !isnum* check is important. The reason for doing this is because the first time the step alias is called, *%context* is set to **\$null**. *%len* > *%context* would always fail if we didn't do the check.

After the aggregating is done, eg. when all rows in a group are processed, mSQLite will call the finalizer alias. The finalizer will receive the *aggregation context* that the step alias used to aggregate data as an argument. The finalizer should then return whatever it wants to return as a final value from the aggregate function. In our example we simply return the max length:

```
Code
alias max_len_finalize {
  var %context = $1
  if (%context !isnum) {
    %context = 0
  }
  return %context
}
```

Once again we check if the aggregation context is number or not. We do this because if there are no rows selected in the query, the step alias is never called, thus *%context* can be **\$null** when finalizer alias is called.

NOTE. As of version 1.0.14 you can specify distinct properties for step and finalize aliases, whereas before you could only create an aggregate with same property for both aliases. This means that you can now write custom aggregates by writing only one alias in mIRC by checking the **\$prop** identifier's value:

```
Code
; An example alias that is used as both step and finalize alias
; You can use this to register the aggregate correspondingly:
; /sqlite_create_aggregate %db max_len max_len max_len 1 step finalize
alias max_len {
  ; Both step and finalize alias needs to check if context isn't set ye
  var %context = $1
  if (%context !isnum) {
    %context = 0
  }
  ; If $prop is step we're in step alias, otherwise in finalize alias
  if ($prop == step) {
    var %len = $2
    if (%len > %context) {
      %context = %len
    }
  }
  ; Again, both step and finalize alias should return the context at th
  return %context
}
```

Aggregate Example

Here's a complete example of implementing the aggregate and example of using it:

Example

```
/*
** mIRC SQLite Demonstration Script
** MAX_LEN Aggregate
**
** Save as sqlite_aggregate.mrc and do
** /load -rs sqlite_aggregate.mrc
**
** This script demonstrates the user
```

```
** defined aggregates of the SQLite.
* *
** Usage:
** /find max len word1 word2 word3 ... wordN
* *
** The /find_max_len will find the maximum
** length of a word from the list of words
** passed to it.
*/
; Our test alias
alias find_max_len {
 var %db = $sqlite_open(words.db)
 if (!%db) {
    echo 4 -a Error: %sqlite_errstr
    return
  }
 ; Do initialize stuff
  sqlite_create_aggregate %db max_len max_len_step max_len_finalize 1
  sqlite_exec %db DROP TABLE IF EXISTS words
  sqlite_exec %db CREATE TABLE words (word)
 var %i = 1
 while (%i <= $0) {</pre>
   var %word = $sqlite_escape_string($gettok($1-, %i, 32))
    sqlite_exec %db INSERT INTO words (word) VALUES (' $+ %word $+ ')
   inc %i
  }
 ; Find the longest length
 var %sql = SELECT max_len(word) FROM words
 var %request = $sqlite_query(%db, %sql)
 ; Aggregate functions always returns something,
  ; so we don't need to check if there are rows
 var %len = $sqlite_fetch_single(%request)
 echo -a The maximum lenght of a word is %len characters.
  sqlite_free %request
  sqlite_close %db
}
; Step alias
alias max_len_step {
 var %context = $1, %len = $len($2)
 if (%context !isnum || %len > %context) {
   %context = %len
  }
return %context
```

```
}
; Finalizer alias
alias max_len_finalize {
  var %context = $1
  if (%context !isnum) {
    %context = 0
  }
  return %context
}
```

Limitations

There are certain limitations for functions and aggregates. These limitations exists in mSQLite because they exist in the underlying software (eg. mIRC and/or SQLite).

- 1. The length of the function name in SQLite can be up to 255 bytes.
- 2. Max number of arguments that is supported is 127.
- 3. Upon executing a query, the user defined function name with all of its arguments can be up to 940 characters. The reason for this is because mIRC can't handle strings longer than this. For most functions this isn't really a problem as 940 characters is a long string. Also note that mSQLite needs to encode special characters of mIRC to ensure that mIRC will properly interpret the function, which can make the query take more characters than expected.
- 4. Lastly, you can't pass binary data to the functions safely, as mIRC isn't binary safe.

Fortunately mSQLite provides you a way around 3 and 4. It defines two SQL helper functions that you can use to pass large amounts of data in a binary-safe manner to user defined functions and back to SQLite:

• TMPARG ([data])

Used to pass large amount of data to an user defined function. mSQLite will write the *data* to a temporary file and then returns the filename to the user defined function. The user defined function can then read and process the contents of the file. The user defined function should not try to delete the file; mSQLite will do so after the query is done executing. If you call this with no arguments an empty temporary file is created.

```
Example
```

```
; process_data is an user defined function that processes bulk of l
var %sql = SELECT process_data(TMPARG(bytes)) FROM binary_data
var %request = $sqlite_query(%db, %sql)
```

• TMPRES (file)

Used to return large amount of data from an user defined function. The user defined function should write the return value to a file and then return the filename. mSQLite will then attempt to open the returned *file*, read its contents and return them. *TMPRES* will automatically try to find out whether the data in the file is text or binary. If you want to enforce it as a certain type use the *CAST* expression. If the user defined function creates

the file that it returns, it is responsible for deleting it. You can use *TMPARG* in conjuction with *TMPRES* by generating an empty temporary file and using it in the user defined function.

Code

; Example 1, using TMPARG to pass data to a function and TMPRES to ; The user defined function can simply write the return value to t ; by TMPARG which was passed to it as first argument and then return sqlite_exec %db UPDATE binary_data SET bytes = TMPRES(process_data)

; Example 2, using TMPARG to generate an empty temporary file and µ ; user defined function, which write data to it and then return the sqlite_exec %db UPDATE binary_data SET bytes = TMPRES(generate_data

; Example 3, not using TMPARG at all, thus leaving the responsibil: ; delete the file used to return data to the user defined function sqlite_exec %db UPDATE binary_data SET bytes = TMPRES(generate_data

sqlite_open

Opens a SQLite database.

```
Syntax
```

```
$sqlite_open ( [ db [, from ] ] )
```

Parameters

db

The filename of the database to open. Optional, see remarks for details.

from

The filename of the database to create memory database from. Optional, see remarks for details.

Return Value

A positive, numeric connection identifier if successful, or **\$null** if there was an error.

Remarks

The *db* argument is optional, if it isn't specified, a transient database is opened in a temporary file. If specified, and the file *db* doesn't exist, an empty database will be created on that file.

If *db* is equal to the special keyword **:memory:** a memory database is opened instead of a file database. If *from* is specified the memory database will contain a copy of the specified database, otherwise an empty memory database is created. If file *from* doesn't exist, an empty database will be created on that file. *from* is only valid when **:memory:** is used, otherwise an error is raised.

If **\$null** is returned you can determine the exact reason for the error by checking the value of **%sqlite_errstr**.

For more information about error handling, see Handling Errors

```
Example
; Opens a database and displays the status after. Closes the db if it w
var %db = $sqlite_open(test.db)
if (%db) {
    echo -a Database opened successfully.
    sqlite_close %db
}
else {
```

```
echo -a Error opening database: %sqlite_errstr
}
```

See Also

sqlite_close sqlite_open_memory sqlite_write_to_file

sqlite_open_memory

Opens a SQLite memory database.

```
Syntax
```

```
$sqlite_open_memory ( [ from ] )
```

Parameters

from

The filename of the database to create memory database from. Optional, see remarks for details.

Return Value

A positive, numeric connection identifier if successful, or **\$null** if there was an error.

Remarks

This function is identical to using <u>\$sqlite_open</u> with the first argument *db* set to special keyword **:memory:**. See its reference for details.

```
Example
; Opens a database and displays the status after. Closes the db if it w
var %db = $sqlite_open_memory()
if (%db) {
    echo -a Memory database created and opened successfully.
    sqlite_close %db
}
else {
    echo -a Error opening a memory database: %sqlite_errstr
}
```

See Also

<u>sqlite_open</u> <u>sqlite_close</u> <u>sqlite_write_to_file</u>

sqlite_write_to_file

Writes a memory database to a file.

```
Syntax
```

```
$sqlite_write_to_file ( conn, file )
/sqlite_write_to_file conn file
```

Parameters

conn The connection identifier of a memory database. *file* The filename to save the memory database to.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

This function allows you to write a memory database to a file. It's useful if you want to work with a database on memory for vastly improved speed, but also keep a permanent copy on disk. Another common use is to debug errors by dumping a memory database to file so you can examine the contents with your favorite visual editor.

sqlite_close

Closes an open SQLite database connection.

```
Syntax
$sqlite_close ( conn )
/sqlite_close conn
```

Parameters

conn The connection identifier.

Return Value

1 if connection was closed successfully, or **\$null** if there was an error.

Remarks

It is usually ok to ignore the return value of \$sqlite_close because the only case an error is returned is when an invalid *conn* is specified.

Example

```
; Opens a database and displays the status after. Closes the db if it w
var %db = $sqlite_open(test.db)
if (%db) {
    echo -a Database opened successfully.
    sqlite_close %db
}
else {
    echo -a Error opening database: %sqlite_errstr
}
```

See Also

sqlite_open

sqlite_query

Executes a SQL query and returns data returned by it.

Syntax

```
$sqlite_query ( conn, query [, bind_value [, ... ] ] ) [ .file ]
/sqlite_query conn query
$sqlite_query ( statement [, bind_value [, ... ] ] )
/sqlite_query statement [ bind_value [ ... ] ]
```

Parameters

conn

The connection identifier.

query

The query to execute.

statement

A prepared statement to execute.

bind_value

Optional. One or more values to bind to the query.

Properties

file

Optional. If specified the query parameter is treated as a filename instead, and that file will be executed as SQL.

Return Value

A positive, numeric result identifier or **0** on success, or **\$null** if there was an error.

Remarks

To execute a prepared statement first prepare it with <u>\$sqlite_prepare</u>. To learn about prepared statements and binding values, see <u>Prepared Statements</u>.

In case of **0** is returned, it means that \$sqlite_query was used to execute a query that doesn't return any data, such as INSERT or UPDATE.

A SELECT query always returns a result identifier on success, even if the query selected no rows. You can use **\$sqlite_num_rows** to determine how many rows were returned.

If **\$null** is returned you can determine the exact reason for the error by checking

the value of **%sqlite_errstr**.

For more information about error handling, see Handling Errors

Note that if you want to bind a text value with more than one word, you must use the identifier form of syntax. If you don't care about the return value, you can use the built-in mIRC command **/noop**

Starting from mSQLite version 1.2.0 you can bind values even for non-prepared statements, but this only works when calling \$sqlite_query as an identifier.

\$sqlite_query can execute multiple queries seperated by semicolons. The
returned result is the data returned by the last SQL query.
To see guidelines for writing SQL queries with mIRC SQLite, see Writing
Queries.

```
Example
; Selects data from a table and fetches it, assumes that a db connectio
var %sql = SELECT col, another FROM table
var %request = $sqlite_query(%db, %sql)
if (%request) {
    echo -a Query executed successfully.
    sqlite_free %request
}
else {
    echo -a Error executing query: %sqlite_errstr
}
```

See Also

Writing Queries Handling Binary Data Prepared Statements sqlite_unbuffered_query sqlite_exec sqlite_fetch_row sqlite_fetch_bound sqlite_fetch_single sqlite_num_rows sqlite_free

sqlite_unbuffered_query

Executes a SQL query and returns data returned by it.

Syntax

```
$sqlite_unbuffered_query ( conn, query [, bind_value [, ... ] ] ) [ .fi]
/sqlite_unbuffered_query conn query
$sqlite_unbuffered_query ( statement [, bind_value [, ... ] ] )
/sqlite_unbuffered_query statement [ bind_value [ ... ] ]
```

Parameters

conn

The connection identifier.

query

The query to execute.

statement

A prepared statement to execute.

bind_value

Optional. One or more values to bind to the query.

Properties

file

Optional. If specified the query parameter is treated as a filename instead, and that file will be executed as SQL.

Return Value

A positive, numeric result identifier or **0** on success, or **\$null** if there was an error.

Remarks

To execute a prepared statement, first prepare it with <u>\$sqlite_prepare</u>. To learn about prepared statements and binding values, see <u>Prepared Statements</u>.

Unbuffered queries work like regular queries, except that they produce a result set that isn't buffered in memory. Since the rows aren't buffered in memory, unbuffered queries are the optimal way to handle large set of sequental data because they're more efficient and the memory footprint is much smaller. The trade off is that you can't random-access the unbuffered result set, like you can with buffered result sets. The following functions aren't supported on unbuffered results: <u>sqlite_num_rows</u>, <u>sqlite_current</u>, <u>sqlite_current_bound</u>, <u>sqlite_result</u>, <u>sqlite_next</u>, <u>sqlite_prev</u>, <u>sqlite_has_more</u>, <u>sqlite_has_prev</u>, sqlite_seek, sqlite_rewind and sqlite_key.

Remember to free the result with <u>sqlite_free</u> when done with the result set. If there's unfetched rows in the result set, and it isn't freed it might block other queries from executing.

In case of **0** is returned, it means that \$sqlite_unbuffered_query was used to execute a query that doesn't return any data, such as INSERT or UPDATE. A SELECT query always returns a result identifier on success, even if the query selected no rows. You can use **\$sqlite_num_rows** to determine how many rows were returned.

If **\$null** is returned you can determine the exact reason for the error by checking the value of **%sqlite_errstr**.

For more information about error handling, see <u>Handling Errors</u>

Note that if you want to bind a text value with more than one word, you must use the identifier form of syntax. If you don't care about the return value, you can use the built-in mIRC command /**noop**

Starting from mSQLite version 1.2.0 you can bind values even for non-prepared statements, but this only works when calling \$sqlite_unbuffered_query as an identifier.

\$sqlite_unbuffered_query can execute multiple queries seperated by semicolons. The returned result is the data returned by the last SQL query. To see guidelines for writing SQL queries with mIRC SQLite, see Writing Queries.

For an example, see <u>\$sqlite_query</u>.

See Also

Writing Queries Handling Binary Data Prepared Statements sqlite_exec sqlite_fetch_row sqlite_fetch_bound sqlite_fetch_single sqlite_num_rows sqlite_free

sqlite_prepare

Prepares a SQL query to be executed later.

Syntax

\$sqlite_prepare (conn, query) [.file]

Parameters

conn The connection identifier. *query*

The query to execute.

Properties

file

Optional. If specified the query parameter is treated as a filename instead, and that file will be executed as SQL.

Return Value

A positive, numeric statement identifier on success, or **\$null** if there was an error.

Remarks

Prepared queries are efficient when you need to execute the same query many times with different parameters. This is because prepared query is only compiled once, and can then be executed without having to re-compile the query. You can bind parameters in prepared queries, for more information about prepared statements and parameter binding, see <u>Prepared Statements</u>. Just like ordinary queries, prepared queries are executed with \$sqlite_exec or \$sqlite_query, see example below.

If **\$null** is returned you can determine the exact reason for the error by checking the value of **%sqlite_errstr**.

For more information about error handling, see <u>Handling Errors</u>.

\$sqlite_prepare can only prepare a single query. Extra queries seperated by a semi-colon are ignored, only the first one is prepared. To see guidelines for writing SQL queries with mIRC SQLite, see <u>Writing</u> Oueries.

```
Example
; Inserts data into a table two times with different parameters
var %sql = INSERT INTO table VALUES (?, :test)
var %stmt = $sqlite_prepare(%db, %sql)
if (%stmt) {
  echo -a Query prepared successfully.
  ; Binds Hello as first parameter, and World as second parameter and i
  sqlite_bind_value %stmt 1 Hello
  sqlite_bind_value %stmt :test World
  sqlite_exec %stmt
  ; Binds NULL as first parameter, and 100 as second parameter and inse
  sqlite_bind_null %stmt 1
  sqlite_bind_value %stmt :test 100
  sqlite_exec %stmt
  ; Binds 'This is a test' as first parameter, and uses the previously
  noop $sqlite_exec(%stmt, This is a test)
  sqlite_free %stmt
}
else {
  echo -a Error preparing query: %sqlite_errstr
}
```

See Also

Prepared Statements sqlite_bind_column sqlite_bind_param sqlite_bind_value sqlite_exec sqlite_query sqlite_free

sqlite_exec

Executes a result-less SQL query.

Syntax

```
$sqlite_exec ( conn, query [, bind_value [, ... ] ] ) [ .file ]
/sqlite_exec conn query
$sqlite_exec ( statement [, bind_value [, ... ] ] )
/sqlite_exec statement [ bind_value [ ... ] ]
```

Parameters

conn

The connection identifier.

query

The query to execute.

statement

A prepared statement to execute.

bind_value

Optional. One or more values to bind to the query.

Properties

file

Optional. If specified the query parameter is treated as a filename instead, and that file will be executed as SQL.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

To execute a prepared statement, first prepare it with <u>\$sqlite_prepare</u>. To learn about prepared statements and binding values, see <u>Prepared Statements</u>.

If **\$null** is returned you can determine the exact reason for the error by checking the value of **%sqlite_errstr**.

For more information about error handling, see Handling Errors

Note that if you want to bind a text value with more than one word, you must use the identifier form of syntax. If you don't care about the return value, you can use the built-in mIRC command /**noop**

Starting from mSQLite version 1.2.0 you can bind values even for non-prepared statements, but this only works when calling \$sqlite_exec as an identifier.

\$sqlite_exec can execute multiple queries seperated by semicolons.
To see guidelines for writing SQL queries with mIRC SQLite, see Writing
Queries.

```
Example
; Inserts data to a table, assumes that a db connection is already esta
var %sql = INSERT INTO table (key, value) VALUES ('version', '1.0.0')
if ($sqlite_exec(%db, %sql)) {
    echo -a Query executed succesfully.
}
else {
    echo -a Error executing query: %sqlite_errstr
}
```

See Also

Writing Queries Handling Binary Data Prepared Statements sqlite_query sqlite_unbuffered_query

sqlite_exec_file

Executes a result-less SQL query from a file.

```
Syntax
$sqlite_exec_file ( conn, file [, bind_value [, ... ] ] )
/sqlite_exec_file conn file
```

Parameters

conn
The connection identifier.
file
The file to execute.
bind_value
Optional. One or more values to bind to the query.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

This is an alias for <a><u>\$sqlite_exec(...).file</u>

This command is useful for executing a long query or multiple queries. One common use is executing an initialization query file after loading a script.

```
Example
; A possible LOAD event for a script
on *:LOAD:{
  var %db = $sqlite_open(script.db)
   sqlite_exec_file %db init.sql
   sqlite_close %db
}
```

See Also

Writing Queries Handling Binary Data Prepared Statements sqlite_exec

sqlite_begin

Begins a transaction.

Syntax
\$sqlite_begin (conn)
/sqlite_begin conn

Parameters

conn The connection identifier.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

\$sqlite_begin is a shorthand function provided for convience for executing *BEGIN TRANSACTION* on *conn*.

Transactions should be used whenever a batch of queries that modify a database are executed. Transactions are much more efficient in such cases, because otherwise every individual query would create a transaction of their own, which is an expensive operation.

It is usually ok to ignore the return value of \$sqlite_begin because the only case an error is returned is when an invalid *conn* is specified or it's used out of sequence.

See Also

sqlite_commit sqlite_rollback sqlite_autocommit

sqlite_commit

Commits a transaction.

Syntax

\$sqlite_commit (conn)
/sqlite_commit conn

Parameters

conn The connection identifier.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

\$sqlite_commit is a shorthand function provided for convience for executing *COMMIT TRANSACTION* on *conn*.

Committing a transaction will save all the changes that were done during the transaction to the database.

It is usually ok to ignore the return value of \$sqlite_commit because the only case an error is returned is when an invalid *conn* is specified or it's used out of sequence.

See Also

<u>sqlite_begin</u> <u>sqlite_rollback</u> <u>sqlite_autocommit</u>

sqlite_rollback

Rolls back a transaction.

Syntax

\$sqlite_rollback (conn)
/sqlite_rollback conn

Parameters

conn The connection identifier.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

\$sqlite_rollback is a shorthand function provided for convience for executing *ROLLBACK TRANSACTION* on *conn*.

Rolling back a transaction will discard all the changes that were done during the transaction.

It is usually ok to ignore the return value of \$sqlite_rollback because the only case an error is returned is when an invalid *conn* is specified or it's used out of sequence.

See Also

<u>sqlite_begin</u> <u>sqlite_commit</u> <u>sqlite_autocommit</u>

sqlite_qt

Add single quotes around text.

```
Syntax
```

```
$sqlite_qt ( text )
```

Parameters

text The text to be single quoted.

Return Value

The single quoted text.

Remarks

This is an auxiliary identifier that can be used to quote data prior to using them in queries.

You should this instead of the mIRC's *\$qt* identifier, the reason is explained in FAQ, under the question "Should I use single or double quotes for string literals?"

Example

```
; Escape data, and then add quotes around it
%data = $sqlite_qt($sqlite_escape_string(%data))
; Execute a query
sqlite_exec %db INSERT INTO table (data) VALUES ( %data )
```

See Also

sqlite_escape_string

sqlite_escape_string

Escapes a string for use as a query parameter.

Syntax

\$sqlite_escape_string (string)

Parameters

string The string to escape.

Return Value

Escaped string.

Remarks

\$sqlite_escape_string escapes the specific *string* so that it can be used safely in queries.

You should always call \$sqlite_escape_string on user input to avoid SQL injection.

Example

```
var %str = $?="Input a string:"
var %sql = INSERT INTO table (value) VALUES (' $+ $sqlite_escape_string
; %sql can now be safely executed
```

See Also

sqlite_query sqlite_exec sqlite_qt

sqlite_create_function

Register an user defined function.

Syntax

\$sqlite_create_function (conn, func_name, func_alias [, num_args [, pr
/sqlite_create_function conn func_name func_alias [num_args [prop]]

Parameters

conn The connection identifier.

func_name

The SQL function name to register.

func_alias

The mIRC alias to register as function.

num_args

The number of arguments the function accepts. Optional.

prop

A custom property you want to use with alias name. Optional.

Return Value

1 if the function was registered successfully, or **\$null** if there was an error.

Remarks

\$sqlite_create_function allows users to register their own functions in SQL. You can also use \$sqlite_create_function to override default functionality of SQLite's core functions.

The optional argument *num_args* can be used to hint SQLite if there's a predetermined amount of arguments.

If an user defined function is used with different parameter count as what was instructed with *num_args*, the query will raise an SQL error.

The default, **-1**, means that an arbitrary number of parameters can be passed to the function.

It is usually ok to ignore the return value of \$sqlite_create_function if you know that the parameters are valid.

For more information about user defined functions, see <u>User Defined Functions</u>.

Example

```
; Returns a specified amount of characters from left side of a md5 gene
alias md5_left {
  var %str = $1, %len = $2
  return $left($md5(%str), %len)
}
; Registers the user defined function in an existing connection %db
sqlite_create_function %db md5_left md5_left 2
; Registers mIRC's $base as convert_base (3 first parameters, no zeropa
sqlite_create_function %db convert_base base 3
; Example queries
; SELECT md5_left(value, 8) FROM strings
; SELECT convert_base(15, 10, 16)
```

See Also

User Defined Functions Function Limitations sqlite_create_aggregate

sqlite_create_aggregate

Register an user defined aggregate.

Syntax

\$sqlite_create_aggregate (conn, func_name, step_alias, finalize_alias
/sqlite_create_aggregate conn func_name step_alias finalize_alias [num

Parameters

conn

The connection identifier.

func_name

The SQL function name to register.

step_alias

The mIRC alias to register as step function.

finalize_alias

The mIRC alias to register as finalize function.

num_args

The number of arguments the function accepts. Optional.

step_prop

A custom property you want to use for step alias. Optional.

finalize_prop

A custom property you want to use for finalize alias. Optional.

Return Value

1 if the function was registered successfully, or **\$null** if there was an error.

Remarks

\$sqlite_create_aggregate allows users to register their own aggregates in SQL. You can also use \$sqlite_create_aggregate to override default functionality of SQLite's core aggregates.

Creating aggregate functions is similiar to creating ordinary functions. The key difference is that \$sqlite_create_aggregate requires two mIRC aliases to be registered:

step_alias is called for each row in a group, it is used to accumulate data. *finalize_alias* is called after all the rows are processed, it is used to return the result to the query.

step_alias and *finalize_alias* always receive an aggregate context as first parameter.

The *step_alias* can use the aggregate context to accumulate data. It should return the aggregate context at end.

When *finalize_alias* is called it can use the aggregate context to determine the accumuluted result.

Parameters passed to the function in the SQL query are passed after the aggregate context.

The optional argument *num_args* can be used to hint SQLite if there's a predetermined amount of arguments.

If an user defined function is used with different parameter count as what was instructed with *num_args*, the query will raise an SQL error.

The default, **-1**, means that an arbitrary number of parameters can be passed to the function.

It is usually ok to ignore the return value of \$sqlite_create_function if you know that the parameters are valid.

For more information about user defined aggregates, see <u>User Defined Functions</u> and <u>User Defined Aggregates</u>.

```
Example
; Creates an aggregate to return the max length of a string
alias max_len_step {
  var %context = $1, %len = $len($2)
  if (%context !isnum || %len > %context) {
    %context = %len
  }
  return %context
}
alias max_len_finalize {
  var %context = $1
  if (%context !isnum) {
    %context = 0
  }
  return %context
}
; Registers the user defined function in an existing connection %db
sqlite_create_aggregate %db max_len max_len_step max_len_finalize
; Example queries
```

```
; SELECT max_len(value) FROM strings
; SELECT max_len(first_name) FROM contacts GROUP BY last_name
```

See Also

User Defined Functions User Defined Aggregates Function Limitations sqlite_create_function

sqlite_signal_error

Allows user defined functions to signal an error.

```
Syntax
$sqlite_signal_error ( error )
/sqlite_signal_error error
```

Parameters

error The error to be signaled.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

It is usually ok to ignore the return value of \$sqlite_signal_error because the only case an error is returned it's called from an alias that isn't an user defined function.

Example

```
; User defined function
alias udf_error {
    sqlite_signal_error this function will always signal an error
}
; Example, always outputs error
sqlite_create_function udf_error udf_error -1
sqlite_exec SELECT udf_error()
if (%sqlite_errno != $SQLITE_OK) {
    echo 4 -a %sqlite_errstr
}
```

See Also

<u>User Defined Functions</u> <u>User Defined Aggregates</u> <u>Function Limitations</u>
sqlite_set_authorizer

Registers an authorizer for a connection.

```
Syntax
$sqlite_set_authorizer ( conn, authorizer_alias [, prop ] )
/sqlite_set_authorizer conn authorizer_alias [ prop ]
```

Parameters

conn The connection identifier. *authorizer_alias* The mIRC alias to register as an authorizer. *prop*

A custom property you want to use with alias name. Optional.

Return Value

1 if the authorizer was registered successfully, or **\$null** if there was an error.

Remarks

An authorizer lets you to examine what kind of query SQLite is trying to run, and change the behavior of the result by either denying the operation, or ignoring it. The foremost purpose of an authorizer is to allow scripts to safely execute user-entered SQL queries, without compromising security of the database.

Only one authorizer can be registered at a time for a connection. Setting a new authorizer for a connection will override the old authorizer, if one is registered. You can unregister the authorizer by passing **\$null** for *authorizer_alias*.

The authorizer should return **\$SQLITE_OK** if the current action should be allowed, **\$SQLITE_DENY** if it should be denied (generates an error) or **\$SQLITE_IGNORE** if it should be ignored (treated as a no-op). You can use an authorizer to make certain columns or functions return NULL by returning **\$SQLITE_IGNORE** for the corresponding action (see below). Returning any other value than the ones previously mentioned will auto-assume **\$SQLITE_OK**.

The authorizer alias will receive a few arguments. The first argument is the type of action SQLite is performing. The second and third argument provide additional information depending on what type of an action is in question. The fourth argument is the name of the database (eg. "main" or "temp") where

applicable. The fifth argument is the name of the inner-most trigger or view that triggered the authorizer, or **\$null** if the authorizer was triggered directly from code.

You can see all the possible types of action that authorizer can be triggered for below, and the associated arguments for it (the 2nd and 3rd arguments of the authorizer alias).

Value	2nd Arg	3rd Arg
1	Index Name	Table Name
2	Table Name	\$null
3	Index Name	Table Name
4	Table Name	\$null
5	Trigger Name	Table name
6	View Name	\$null
7	Trigger name	Table Name
8	View Name	\$null
9	Table Name	\$null
10	Index Name	Table Name
11	Table Name	\$null
12	Index Name	Table Name
13	Table Name	\$null
14	Trigger Name	Table Name
15	View Name	\$null
16	Trigger Name	Table Name
17	View Name	\$null
18	Table Name	\$null
19	Pragma Name	1st Arg or \$
20	Table Name	Column Name
21	\$null	\$null
22	\$null	\$null
23	Table Name	Column Name
24	Filename	\$null
25	Database Name	\$null
26	Database Name	Table Name
27	Index Name	\$null
28	Table Name	\$null
29	Table Name	Module Name
30	Table Name	Module Name
31	Function Name	\$null
	Value 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	Value2nd Arg1Index Name2Table Name3Index Name4Table Name5Trigger Name6View Name7Trigger name8View Name9Table Name10Index Name11Table Name12Index Name13Table Name14Trigger Name15View Name16Trigger Name17View Name18Table Name19Pragma Name20Table Name21\$null23Table Name24Filename25Database Name26Database Name27Index Name28Table Name29Table Name30Table Name31Function Name

Example

; The following example demonstrates how to create an authorizer that: ; 1) Disallows queries of other type than SELECT. ; 2) Disallows access to certain columns by making SQLite return NULL f alias my_authorizer {

```
; To make the function look cleaner, let's assign the tokens to varia
 var %type = $1
  ; First deny queries other than SELECT
 if (%type != $SQLITE_SELECT && %type != $SQLITE_READ && %type != $SQL
   ; Not authorized!
   return $SQLITE_DENY
 }
 ; Next make sure that we aren't trying to access a "secret" column, e
 if (%type == $SQLITE_READ) {
   var %table = $2, %column = $3
   if (%table == users && %column == password) {
      ; By returning $SQLITE_IGNORE SQLite will return NULL for this co
      ; of denying the whole query, because in that case query such as
      return $SQLITE_IGNORE
   }
 }
 ; Nothing special in this action, allow it.
 return $SQLITE_OK
}
; To register the authorizer do (assumes that %db exists):
sqlite_set_authorizer %db my_authorizer
```

sqlite_free

Frees a query result or prepared statement.

Syntax

```
$sqlite_free ( result )
/sqlite_free result
$sqlite_free ( statement )
/sqlite_free statement
```

Parameters

result The result identifier. *statement*

The statement identifier.

Return Value

1 if the result was freed successfully, or **\$null** if there was an error.

Remarks

It is usually ok to ignore the return value of \$sqlite_free because the only case an error is returned is when an invalid *result* is specified.

```
Example
```

```
; Selects data from a table and then frees it (unpractical, only shows
var %sql = SELECT * FROM table
var %request = $sqlite_query(%db, %sql)
if (%request) {
    echo -a Query executed succesfully. Freeing data.
    sqlite_free %request
}
else {
    echo -a Error executing query: %sqlite_errstr
}
```

See Also

<u>sqlite_query</u> <u>sqlite_finalize</u>

sqlite_finalize

Frees a query result or prepared statement.

Syntax

```
$sqlite_finalize ( result )
/sqlite_finalize result
$sqlite_finalize ( statement )
/sqlite_finalize statement
```

Parameters

result The result identifier.

statement The statement identifier.

Return Value

1 if the result was freed successfully, or **\$null** if there was an error.

Remarks

This is an alias for <u>\$sqlite_free</u> provided for convience for those, who prefer to use finalize when freeing prepared statements.

See Also

<u>sqlite_query</u> <u>sqlite_free</u>

sqlite_fetch_row

Fetches the current row from a result and then advances to the next row.

Syntax

\$sqlite_fetch_row (result, hash_table [, result_type])

Parameters

result The result identifier.

hash_table

The name of the hash table to where to store the row data.

result_type

The type of the result. Optional, see remarks for more info.

Return Value

1 on success; Otherwise **0** if there are no more rows available, or **\$null** if there was an error.

Remarks

\$sqlite_fetch_row fetches the next row from the result and stores the data in
hash_table.

If the hash table doesn't exist, it will be created; Otherwise it will be cleared before new data is stored.

result_type specifies how the hash table is created, it can be one of the following: **\$SQLITE_NUM**, **\$SQLITE_ASSOC** or **\$SQLITE_BOTH**. **\$SQLITE_BOTH** is default.

If \$SQLITE_NUM is used, the hash table items will be field indexes, starting from index 1. If \$SQLITE_ASSOC is used, the items will be field names. If \$SQLITE_BOTH is used, both column indexes and names are used. In case of \$SQLITE_BOTH, if some of the column names are identical to another columns' index, the index has priority and will be used as an item.

```
Example
```

```
; This code assumes a connection is already established and stored in %
var %sql = SELECT first_name, last_name FROM contacts
var %res = $sqlite_query(%db, %sql)
if (%res) {
    echo -a Fetching results...
    echo -a -
```

```
while ($sqlite_fetch_row(%res, row, $SQLITE_ASSOC)) {
    ; If you used $SQLITE_FETCH_NUM or $SQLITE_FETCH_BOTH you could use
    echo -a First name: $hget(row, first_name)
    echo -a Last name: $hget(row, last_name)
    echo -a -
    }
    sqlite_free %res
}
else {
    echo -a Error executing query: %sqlite_errstr
}
```

See Also

sqlite_query
sqlite_fetch_single

sqlite_fetch_bound

Fetches the current row from a result and assigns the column values in variables and then advances to the next row.

Syntax
\$sqlite_fetch_bound (result [, bind_type])

Parameters

result The result identifier. *bind_type* The type of the bind. Optional, see remarks for more info.

Return Value

1 on success; Otherwise **0** if there are no more rows available, or **\$null** if there was an error.

Remarks

\$sqlite_fetch_bound fetches the next row from the *result* and assigns the column data in variables specified by <u>\$sqlite_fetch_field</u>.

bind_type specifies how the values are bound, it can be one of the following: **\$SQLITE_ALL** or **\$SQLITE_BOUND**. **\$**SQLITE_BOUND is default. If **\$**SQLITE_BOUND is specified, only columns that have been bound with <u>\$sqlite_fetch_field</u> are fetched in variables. If **\$**SQLITE_ALL is specified all rows are fetched, even ones that haven't been bound explicitly with <u>\$sqlite_fetch_field</u>. In this case the column names are used as variable names. Depending on whether the column type is binary or not, a regular variable or a binary variable will be used.

The bound variables are set as global variables when fetched, because mSQLite has no access to local variables. You should be very careful that you don't override any existing global variables, especially when \$SQLITE_ALL is used!

```
Example
```

```
; This code assumes a connection is already established and stored in %
var %sql = SELECT first_name, last_name, address FROM contacts
var %res = $sqlite_query(%db, %sql)
if (%res) {
   ; The first column will be bound to %name
```

```
sqlite_bind_field %result 1 name
  ; The third column will be bound to %postal_address
  sqlite_bind_field %result address first_name
  ; The second column will be bound automatically to %last_name with $S
  echo -a Fetching results...
  echo -a -
  while ($sqlite_fetch_bound(%res, $SQLITE_ALL)) {
    ; If you used $SQLITE_BOUND, %last_name would not exist because it
    echo -a First name: %name
    echo -a Last name: %last_name
    echo -a Address: %postal_address
    echo -a -
  }
  sqlite_free %res
}
else {
  echo -a Error executing query: %sqlite_errstr
}
```

See Also

<u>sqlite_query</u> <u>sqlite_bind_field</u> <u>sqlite_fetch_row</u> <u>sqlite_fetch_single</u>

sqlite_fetch_single

Fetches and returns the first column of the current row from a result and then advances to the next row.

Syntax
\$sqlite_fetch_single (result [, &binvar;])

Parameters

result The result identifier.

binvar

The name of the binary variable to assign binary data to. Optional.

Return Value

The value of the first column of the fetched row if *&binvar*; isn't specified, otherwise the size of the binary variable on success, **\$null** if there are no more rows, or if there was an error.

Remarks

If *&binvar*; is specified the behaviour of \$sqlite_fetch_single changes slightly. Instead of returning the first column's value, it will assign it to a binvar and return the binvar's size on success.

In case the first column is not blob type, its text representation will be stored in the *&binvar*; as sequential ascii values. If *&binvar*; isn't set, but the first column is a blob, it will be converted to text.

For more information about handling binary data in mSQLite, see <u>Handling</u> <u>Binary Data</u>

In case of **\$null** is returned it can mean three different things:

- **1.** The returned value from SQLite database is NULL.
- 2. There are no more rows available.
- **3.** There was an error.

To determine the cause of \$null, examine the **%sqlite_errno** variable after calling \$sqlite_fetch_single. The returned value can be one of the following:

1. \$SQLITE_OK if there was no error.

2. **\$SQLITE_NOMOREROWS** if there are no more rows available.

3. Some other of the Error Codes if there was an error.

```
Example
; This code assumes a connection is already established and stored in %
var %sql = SELECT COUNT(*) FROM contacts
var %res = $sqlite_query(%db, %sql)
if (%res) {
    echo -a Number of rows in contacts: $sqlite_fetch_single(%res)
    sqlite_free %res
}
else {
    echo -a Error executing query: %sqlite_errstr
}
```

See Also

sqlite_query sqlite_fetch_row sqlite_fetch_field

sqlite_fetch_field

Fetches and returns the specified column of the current row from a result and then advances to the next row.

Syntax

\$sqlite_fetch_field (result, field [, &binvar;]) [.name]

Parameters

result

The result identifier.

field

The field index or name. See remarks for details.

binvar

The name of the binary variable to assign binary data to. Optional.

Properties

name Forces field to be treated as name.

Return Value

The value of the specified column of the fetched row if *&binvar;* isn't specified, otherwise the size of the binary variable on success, **\$null** if there are no more rows, or if there was an error.

Remarks

\$sqlite_fetch_field is identical to \$sqlite_fetch_single with the only difference being that \$sqlite_fetch_field returns a value of specified column, instead of the first column.

If *field* is numeric it is treated as an ordinal index for the column, first column being 1, otherwise it is treated as the column's name. You can use the **.name** property to force the field to be treated as column name even if it's a number.

See <u>\$sqlite_fetch_single</u> for more details.

```
Example
; This code assumes a connection is already established and stored in %
var %sql = SELECT 1, 'test' AS '1'
var %res = $sqlite_query(%db, %sql)
if (%res) {
```

```
; This treats 1 as field ordinal, thus first column is returned
echo -a Fetch 1st field: $sqlite_fetch_field(%res, 1)
; Go back to previous row because $sqlite_fetch_field increments row
sqlite_rewind %res
; This treats 1 as field nane because of the .name property, second c
echo -a Fetch 2nd field: $sqlite_fetch_field(%res, 1).name
; Free result
sqlite_free %res
}
else {
   echo -a Error executing query: %sqlite_errstr
}
; Output:
; Fetch 1st field: 1
; Fetch 2nd field: a
```

See Also

sqlite_query sqlite_fetch_row sqlite_fetch_single sqlite_result

sqlite_fetch_all

Fetches everything into a file.

Syntax
\$sqlite_fetch_all (result, file [, delim = 9])

Parameters

result The result identifier. **file** The output filename. **delim** Optional. Delimiter in ASCII used to separate fields.

Return Value

1 on success or **\$null** on error.

Remarks

\$sqlite_fetch_all is useful for fetching everything into a single file if you wish to process it using a command such as */filter* or */play* through custom alias.

Each line in the resulting fill will consist of a single row. All fields in the line are separated by **delim**, which is TAB by default. You can specify your own delimiter.

Because it's possible that the data for a field in a row can consist of unsafe characters that would mess up the rows/fields, mIRC SQLite encodes the special characters in the resulting file.

The characters that are encoded are: $\ (backslash), \ (newline), \ (carriage return), \ (null-byte, in binary data) and whatever delimiter is used. The characters are encoded as an escape sequence <math>\ NN$ where NN is a two-digit hexadecimal number.

You can decode the data with <u>\$sqlite_safe_decode</u> if you need to.

See Also

sqlite_query sqlite_fetch_row sqlite_fetch_single sqlite_fetch_field sqlite_safe_encode sqlite_safe_decode

sqlite_current

Fetches the current row from a result.

Syntax

\$sqlite_current (result, hash_table [, result_type])

Parameters

result The result identifier.

hash_table

The name of the hash table to where to store the row data.

result_type

The type of the result. Optional, see remarks for more info.

Return Value

1 on success; Otherwise **0** if the current row position is beyond final row, or **\$null** if there was an error.

Remarks

\$sqlite_current is identical to <u>\$sqlite_fetch_row</u> except it doesn't advance to the
next row.

See Also sqlite fetch row

sqlite_current_bound

Fetches the current row from a result and assigns the column values in variables.

Syntax

\$sqlite_current_bound (result [, bind_type])

Parameters

result The result identifier.

bind_type

The type of the bind. Optional, see remarks for more info.

Return Value

1 on success; Otherwise **0** if the current row position is beyond final row, or **\$null** if there was an error.

Remarks

\$sqlite_current_bound is identical to <u>\$sqlite_fetch_bound</u> except it doesn't
advance to the next row.

See Also

sqlite_fetch_bound

sqlite_result

Fetches and returns the specified column of the current row from a result.

Syntax
\$sqlite_result (result, field [, &binvar;]) [.name]

Parameters

result The result identifier.

field

The field index or name. See remarks for details.

binvar

The name of the binary variable to assign binary data to. Optional.

Properties

name Forces field to be treated as name.

Return Value

The value of the specified column of the fetched row if *&binvar*; isn't specified, otherwise the size of the binary variable on success, **\$null** if there are no more rows, or if there was an error.

Remarks

\$sqlite_result is identical to <u>\$sqlite_fetch_field</u> except it doesn't advance to the next row.

See Also

sqlite_fetch_field

sqlite_num_rows

Returns a number of rows in a result.

Syntax

\$sqlite_num_rows (result)

Parameters

result The result identifier.

Return Value

The number of rows in the result on success, or **\$null** if there was an error.

```
Example
; This code assumes a connection is already established and stored in %
var %sql = SELECT * FROM table
var %res = $sqlite_query(%db, %sql)
if (%res) {
    echo -a Number of rows returned: $sqlite_num_rows(%res)
    sqlite_free %res
}
else {
    echo -a Error executing query: %sqlite_errstr
}
```

See Also

<u>sqlite_query</u> <u>sqlite_num_fields</u>

sqlite_num_fields

Returns a number of fields in a result.

Syntax

```
$sqlite_num_fields ( result )
```

Parameters

result The result identifier.

Return Value

The number of fields in the result on success, or **\$null** if there was an error.

```
Example
; This code assumes a connection is already established and stored in %
var %sql = SELECT * FROM table
var %res = $sqlite_query(%db, %sql)
if (%res) {
    echo -a Number of fields returned: $sqlite_num_fields(%res)
    sqlite_free %res
}
else {
    echo -a Error executing query: %sqlite_errstr
}
```

See Also

sqlite_query sqlite_num_rows

sqlite_changes

Returns a number of affected rows of the last INSERT, UPDATE or DELETE query.

Syntax
\$sqlite_changes (conn)

Parameters

conn The connection identifier.

Return Value

The number of affected rows on success, or **\$null** if there was an error.

```
Example
; This code assumes a connection is already established and stored in %
var %sql = UPDATE publishers SET publisher = 'Square Enix' WHERE publis
if ($sqlite_exec(%db, %sql)) {
    echo -a Number of rows affected: $sqlite_changes(%db)
    sqlite_free %res
}
else {
    echo -a Error executing query: %sqlite_errstr
}
```

See Also

<u>sqlite_exec</u> <u>sqlite_query</u>

sqlite_last_insert_rowid

Returns the row id of the most recently inserted row.

```
Syntax
```

\$sqlite_last_insert_rowid (conn)

Parameters

conn The connection identifier.

Return Value

The row id on success, or **\$null** if there was an error.

```
Example
; This code assumes a connection is already established and stored in %
var %sql = INSERT INTO publishers (publisher) VALUES ('Square Enix')
if ($sqlite_exec(%db, %sql)) {
   echo -a Insrted row id: $sqlite_last_insert_rowid(%db)
   sqlite_free %res
}
else {
   echo -a Error executing query: %sqlite_errstr
}
```

See Also

<u>sqlite_exec</u> <u>sqlite_query</u>

sqlite_field_name

Returns the name of the specified field.

Syntax

\$sqlite_field_name (result, field_index)

Parameters

result The result identifier. *field_index* The ordinal index of the field. The first field has an index of 1.

Return Value

The name of the field on success, or **\$null** if there was an error.

Example

```
; This code assumes a connection is already established and stored in %
var %sql = SELECT * FROM table
var %res = $sqlite_query(%db, %sql)
if (%res) {
    echo -a Name of the first field: $sqlite_field_name(%res, 1)
    sqlite_free %res
}
else {
    echo -a Error executing query: %sqlite_errstr
}
```

See Also

sqlite_query

sqlite_field_type

Returns the type of the specified field.

Syntax

\$sqlite_field_type (result, field_index [, row_index])

Parameters

result The result identifier.

field_index

The ordinal index of the field. The first field has an index of 1.

row_index

The row index to get the type for. Optional; If result is unbuffered or if omitted current row is assumed.

Return Value

The type of the field on success, or **\$null** if there was an error.

Remarks

The returned type can be one of the following values: **\$SQLITE_INTEGER**, **\$SQLITE_FLOAT**, **\$SQLITE_TEXT**, **\$SQLITE_BLOB** and **\$SQLITE_NULL**.

SQLite uses so called manifest typing, which means that field types aren't static. This means that every row can store any type for any field. \$sqlite_field_type can be used to determine the type of the field for a specified row. There are exceptions to this, for details see <u>Datatypes In SQLite Version 3</u>.

See Also

sqlite_query

sqlite_bind_column

Binds a column to a variable.

Syntax

```
$sqlite_bind_column ( result, column, var ) [ .name ]
/sqlite_bind_column result column var
```

Parameters

result The result identifier.

column

The column number of name to bind for. Must exist in the result set. *var* The variable or binary variable to bind the column for.

Properties

name Forces column to be treated as name.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

This is an alias for <u>\$sqlite_bind_field</u> provided for convience.

See Also

Prepared Statements sqlite_bind_field sqlite_prepare sqlite_fetch_bound

sqlite_bind_field

Binds a column to a variable.

Syntax

```
$sqlite_bind_field ( result, column, var ) [ .name ]
/sqlite_bind_field result column var
```

Parameters

result The result identifier.

column

The column number of name to bind for. Must exist in the result set.

var

The variable or binary variable to bind the column for.

Properties

name Forces column to be treated as name.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

The columns bound to variables with \$sqlite_bind_field are used when fetching rows with **\$sqlite_fetch_bound** or **\$sqlite_current_bound**.

If *column* is numeric it is treated as an ordinal index for the column, first column being 1, otherwise it is treated as the column's name. You can use the **.name** property to force the field to be treated as column name even if it's a number.

The *var* parameter is considered as a binary variable if it starts with a **&**. Otherwise it's considered as a regular variable. You should **not** prefix the var with a **%**; otherwise mIRC will evaluate the variable right away.

The bound variables are set as global variables when fetched, because mSQLite has no access to local variables. You should be very careful that you don't override any existing global variables.

For more information about parameter binding, see <u>Prepared Statements</u>.

If you want to use the **.name** property to force the *column* to act as a column name, you must use the first form of the syntax. If you don't care about the return value, you can use the mIRC's built-in command **/noop**

See Also

Prepared Statements sqlite_bind_column sqlite_prepare sqlite_fetch_bound sqlite_current_bound

sqlite_bind_param

Binds a variable as a parameter for prepared statement.

```
Syntax
$sqlite_bind_param ( statement, param, var [, datatype ] )
/sqlite_bind_param statement param var [ datatype ]
```

Parameters

statement

The prepared statement identifier.

param

The parameter to bind to. Must exist in the prepared query.

var

The variable or binary variable to bind to.

datatype

Optional. Tells what datatype var is. See remarks.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

\$sqlite_bind_param can be used to bind a variable to a parameter. The variable is bound as a reference and is evaluated at the time of execution. This means that by changing the variable in mIRC, you're effectively changing the bound value as well.

The *param* parameter can either be a numerical index, specified with a **?** in the query, or a named parameter specified with a **:name** in the query. If binding a named parameter, you should also include the colon in the name.

The *var* parameter is considered as a binary variable if it starts with a **&**. Otherwise it's considered as a regular variable. You should **not** prefix the var with a **%**; otherwise mIRC will evaluate the variable right away. If not a binary variable, the specified variable must be a global variable because local variables only exist in scope of the alias they're declared in; mSQLite has no access to them. See example below.

If *datatype* is specified, it must be one of the **\$SQLITE_INTEGER**, **\$SQLITE_FLOAT**, **\$SQLITE_TEXT**, **\$SQLITE_BLOB** and

\$SQLITE_NULL. The datatype tells what datatype var is. If omitted, mSQLite will attempt to deduce the datatype of the variable at execution time. You should specify datatype when you want a numerical variable to act as a text for example.

For more information about parameter binding, see Prepared Statements.

```
Example
; Open a temporary db
var %db = $sqlite_open()
; Binds one numerical and one named parameter two times
var %sql = SELECT ?, :test
var %stmt = $sqlite_prepare(%db, %sql)
if (%stmt) {
  ; Binds %first as first parameter, and &second; as second parameter.
  ; Do not prefix the variable with a % or mIRC will evaluate the varia
  sqlite_bind_param %stmt 1 first
  ; If datatype isn't specified the binary variable would be considered
  sqlite_bind_param %stmt :test &second; $SQLITE_TEXT
  ; We can declare the variables after they're bound because they aren'
  set %first Hello
  bset -t &second; 1 World
  ; Execute the guery and show the results
  var %result = $sqlite_query(%stmt)
  if ($sqlite_fetch_row(%result, row, $SQLITE_NUM)) {
    echo -a First execution:
    echo -a 1st: $hget(row, 1)
    echo -a 2nd: $hget(row, 2)
  }
  sqlite_free %result
  ; Change the first parameter to something else, you don't need to cal
  set %first Another
  ; Execute the query again and show the new results
  var %result = $sqlite_query(%stmt)
  if ($sqlite_fetch_row(%result, row, $SQLITE_NUM)) {
    echo -a Second execution:
    echo -a 1st: $hget(row, 1)
    echo -a 2nd: $hget(row, 2)
  }
  sqlite_free %result
  sqlite_free %stmt
}
```

```
else {
   echo -a Error preparing query: %sqlite_errstr
}
sqlite_close %db
; Output:
; First execution:
; 1st: Hello
; 2nd: World
; Second execution:
; 1st: Another
; 2nd: World
```

See Also

<u>Prepared Statements</u> <u>sqlite_prepare</u> <u>sqlite_bind_field</u> <u>sqlite_bind_value</u>

sqlite_bind_value

Binds a value as a parameter for prepared statement.

```
Syntax
$sqlite_bind_value ( statement, param, value [, datatype ] )
/sqlite_bind_value statement param value [ datatype ]
```

Parameters

statement The prepared statement identifier.

param
The parameter to bind to. Must exist in the prepared query.
value
The value to bind to.
datatype
Optional. Tells what datatype value is. See remarks.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

\$sqlite_bind_value can be used to bind a value to a parameter.

The *param* parameter can either be a numerical index, specified with a **?** in the query, or a named parameter specified with a **:name** in the query. If binding a named parameter, you should also include the colon in the name.

If *datatype* is specified, it must be one of the **\$SQLITE_INTEGER**, **\$SQLITE_FLOAT**, **\$SQLITE_TEXT**, **\$SQLITE_BLOB** and **\$SQLITE_NULL**. The datatype tells what datatype value is. If omitted, mSQLite will attempt to deduce the datatype of the value. You should specify datatype when you want a numerical value to act as a text for example.

Note that if you want to bind a text value with more than one word, you must use the first form of syntax. If you don't care about the return value, you can use the built-in mIRC command **/noop**

For more information about parameter binding, see <u>Prepared Statements</u>.

Example

```
; Open a temporary db
var %db = $sqlite_open()
; Binds one numerical and one named parameter
var %sql = SELECT ?, :test
var %stmt = $sqlite_prepare(%db, %sql)
if (%stmt) {
  ; Binds 'Hello world' as first parameter and 100 as second parameter
  ; We must use the $sqlite_bind_param syntax here, because the value c
  noop $sqlite_bind_value(%stmt, 1, Hello world)
  ; If datatype isn't specified 100 would be considered an integer by m
  sqlite_bind_value %stmt :test 100 $SQLITE_FLOAT
  ; Execute the query and show the results
  var %result = $sqlite_query(%stmt)
  if ($sqlite_fetch_row(%result, row, $SQLITE_NUM)) {
    echo -a 1st: $hget(row, 1)
    echo -a 2nd: $hget(row, 2)
  }
  sqlite_free %result
  sqlite_free %stmt
}
else {
  echo -a Error preparing query: %sqlite_errstr
}
sqlite_close %db
; Output:
; 1st: Hello world
; 2nd: 100.0
```

See Also

Prepared Statements sqlite_prepare sqlite_bind_field sqlite_bind_param

sqlite_clear_bindings

Clears all bindings from a result or a statement.

```
Syntax
$sqlite_clear_bindings ( result )
/sqlite_clear_bindings result
$sqlite_clear_bindings ( statement )
/sqlite_clear_bindings statement
```

Parameters

result The result identifier. *statement*

The prepared statement identifier.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

\$sqlite_clear_bindings clear all parameter bindings from a specified result or prepared statement. That is, unless they're re-bound, they will default to NULL.

If used to clear bindings in a result set, clears all bindings specified with \$sqlite_bind_field. If used to clear bindings in a prepared statement, clears all bindings specified with \$sqlite_bind_param or \$sqlite_bind_value.

It is usually ok to ignore the return value of \$sqlite_clear_bindings because the only case an error is returned is when an invalid *statement* is specified.

See Also

Prepared Statements sqlite_prepare sqlite_bind_param sqlite_bind_value

sqlite_next

Seek to the next row number.

Syntax

\$sqlite_next (result)
/sqlite_next result

Parameters

result The result identifier.

Return Value

1 on success; Otherwise **0** if there are no more rows available, or **\$null** if there was an error.

See Also

sqlite_query sqlite_num_rows sqlite_prev sqlite_has_more sqlite_has_prev

sqlite_prev

Seek to the previous row number.

Syntax

\$sqlite_prev (result)
/sqlite_prev result

Parameters

result The result identifier.

Return Value

1 on success; Otherwise **0** if there is no previous row available, or **\$null** if there was an error.

See Also

sqlite_query sqlite_num_rows sqlite_next sqlite_has_more sqlite_has_prev

sqlite_has_more

Returns whether or not more rows are available.

```
Syntax
$sqlite_has_more ( result )
```

Parameters

result The result identifier.

Return Value

1 if there are more rows available; Otherwise **0** if there are no more rows available, or **\$null** if there was an error.

```
Example
; This code assumes a connection is already established and stored in %
var %sql = SELECT * FROM table
var %res = $sglite_guery(%db, %sgl)
if (%res) {
  ; Counts number of rows, unpractical as you could just use $sqlite_nu
  var %rows = 0
  while ($sqlite_has_more(%res)) {
    sqlite_next %res
    inc %rows
  }
  echo -a Number of rows returned: %rows
  sqlite_free %res
}
else {
  echo -a Error executing guery: %sglite_errstr
}
```

See Also

sqlite_query sqlite_num_rows sqlite_next sqlite_prev sqlite_has_prev
sqlite_has_prev

Returns whether or not a previous row is available.

```
Syntax
$sqlite_has_prev ( result )
```

Parameters

result The result identifier.

Return Value

1 if there is a previous available; Otherwise **0** if there is no previous row available, or **\$null** if there was an error.

See Also

sqlite_query sqlite_num_rows sqlite_prev sqlite_next sqlite_has_more

sqlite_seek

Seek to a particular row.

Syntax

```
$sqlite_seek ( result, row_index [, seek_type ] )
/sqlite_seek result row_index [ seek_type ]
```

Parameters

result The result identifier.

row_index

The row to seek to.

seek_type

The seek type. Optional, see remarks for more info.

Return Value

1 on success; Otherwise **0** if the row isn't seekable, or **\$null** if there was an error.

Remarks

The optional *seek_type* parameter specifies the direction and offset of seek and can be one of the following: **\$SQLITE_BEG**, **\$SQLITE_CUR** or **\$SQLITE_END**. **\$**SQLITE_BEG is default.

\$SQLITE_BEG seeks forward from the start of the result. \$SQLITE_CUR seeks forward from the current position of the result. \$SQLITE_END seeks backwards from the end of the result.

See Also

<u>sqlite_query</u> <u>sqlite_key</u>

sqlite_rewind

Seeks to the first row.

Syntax

\$sqlite_rewind (result)
/sqlite_rewind result

Parameters

result The result identifier.

Return Value

1 on success; Otherwise **0** if the result isn't rewindable, or **\$null** if there was an error.

Remarks

\$sqlite_rewind is equivalent to calling \$sqlite_seek with row value of 1 and seek_type of \$SQLITE_BEG.

See Also

sqlite_query sqlite_key sqlite_rewind

sqlite_key

Returns the current row number of a result.

```
Syntax
$sqlite_key ( result )
```

Parameters

result The result identifier.

Return Value

The current row number on success, or **\$null** if there was an error.

Remarks

\$sqlite_rewind is equivalent to calling \$sqlite_seek with row value of 1 and seek_type of \$SQLITE_BEG.

See Also

sqlite_query sqlite_key sqlite_rewind

sqlite_help

Opens the help file.

Syntax

/sqlite_help

sqlite_libversion

Returns the version of the SQLite library.

```
Syntax
$sqlite_libversion
```

Return Value

The version of the library.

Remarks

The returned version is delimited by periods and has 3 different numbers indicating the version: a **major**, a **minor** and a **revision** number. For example 1.2.3 means that major version is 1, minor 2 and the revision number is 3.

Example

```
; Displays the SQLite version to an active window
//echo -a SQLite Version: $sqlite_libversion
; Example output:
; SQLite Version: 3.3.6
```

See Also

sqlite_dllversion

sqlite_dllversion

Returns the version of the mIRC SQLite DLL.

```
Syntax
$sqlite_dllversion
```

Return Value

The version of the library.

Remarks

The returned version is delimited by periods and has 3 different numbers indicating the version: a *major*, a *minor* and a *revision* number. For example 1.2.3 means that major version is 1, minor 2 and the revision number is 3.

Example

```
; Displays the DLL version to an active window
//echo -a DLL Version: $sqlite_dllversion
; Example output:
; DLL Version: 1.0.0
```

See Also

sqlite_libversion

sqlite_error_string

Returns a textual representation of an error code.

```
Syntax
```

```
$sqlite_error_string ( errcode )
```

Parameters

errcode The error code to format.

Return Value

A formatted, textual representation of *errcode*.

Example

```
; Displays textual representation of the latest error code which is sto
var %err = %sqlite_errno
echo -a Last Error: $sqlite_error_string(%err)
; Example output:
; Last Error: not an error
```

See Also

sqlite_last_error

sqlite_busy_timeout

Sets the busy timeout duration or disables busy handlers.

Syntax

```
$sqlite_busy_timeout ( conn, milliseconds )
/sqlite_busy_timeout conn milliseconds
```

Parameters

conn The connection identifier. *milliseconds* The number of milliseconds.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

Set the maximum time, in milliseconds, that SQLite will wait for a *conn* to become ready for use.

Use **0** for *milliseconds* to disable busy handlers.

You can set the default busy timeout in the mSQLite configuration file, see <u>Configuring mIRC SQLite</u> for details.

It is usually ok to ignore the return value of \$sqlite_busy_timeout because the only case an error is returned is when an invalid *conn* is specified.

sqlite_is_valid_conn

Checks if a database connection is valid.

```
Syntax
$sqlite_is_valid_conn ( conn )
```

Parameters

conn The connection identifier.

Return Value

1 if *conn* is a valid connection, **0** if it's invalid, or **\$null** if there was an error.

Remarks

It is usually ok to ignore if \$sqlite_is_valid_conn returns \$null because the only case an error is returned is when *conn* isn't specified.

See Also

<u>sqlite_open</u> <u>sqlite_is_valid_result</u> <u>sqlite_is_valid_statement</u> <u>sqlite_is_memory</u>

sqlite_is_valid_result

Checks if a result is valid.

Syntax
\$sqlite_is_valid_result (result)

Parameters

result The result identifier.

Return Value

1 if *result* is a valid result, **0** if it's invalid, or **\$null** if there was an error.

Remarks

It is usually ok to ignore if \$sqlite_is_valid_result returns \$null because the only case an error is returned is when *result* isn't specified.

See Also

<u>sqlite_query</u> <u>sqlite_is_valid_conn</u> <u>sqlite_is_valid_statement</u> <u>sqlite_is_memory</u>

sqlite_is_valid_statement

Checks if a prepared statement is valid.

Syntax
\$sqlite_is_valid_statement (statement)

Parameters

result The statement identifier.

Return Value

1 if *statement* is a valid prepared statement, **0** if it's invalid, or **\$null** if there was an error.

Remarks

It is usually ok to ignore if \$sqlite_is_valid_statement returns \$null because the only case an error is returned is when *statement* isn't specified.

See Also

sqlite_prepare sqlite_is_valid_conn sqlite_is_valid_result sqlite_is_memory

sqlite_is_memory

Checks whether database is memory database or not.

Syntax

\$sqlite_is_memory (conn)

Parameters

conn The connection identifier.

Return Value

1 if *conn* is a memory database, **0** if it's file database, or **\$null** if there was an error.

See Also

<u>sqlite_is_valid_conn</u> <u>sqlite_is_valid_result</u>

sqlite_field_metadata

Retrieves meta information about a specific column.

Syntax

\$sqlite_field_metadata (conn [, database], table, column, htable)

Parameters

conn

The connection identifier.

database

The database name where table exists. Optional, see remarks for details.

table

The table name where column exists.

column

The column to get meta information about.

htable

The hash table to assign results to.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

This identifier fetches meta information about a specific column, and assigns the results to *htable*.

The hash table will have the following items set:

[pre]

dattype The declared data type, eg. INTEGER

collseq The collation sequence name, eg. *BINARY*

notnull 1 if *NOT NULL* constraint exists, otherwise 0

primkey 1 if column is part of *PRIMARY KEY*, otherwise 0

autoinc 1 if column is *AUTOINCREMENT*, otherwise 0

[/pre]

The *database* argument is optional. If omitted all attached databases in *conn* will be searched for the specified table.

sqlite_load_extension

Loads a dynamic extension.

Syntax

```
$sqlite_load_extension ( conn, filename [, entrypoint ] )
/sqlite_load_extension conn filename [ entrypoint ]
```

Parameters

conn The connection identifier. *filename* The filename of the extension. *entrypoint*

The entrypoint of the extension Optional, see remarks for details.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

This function loads a dynamic extension *filename*. Dynamic extensions are useful for creating efficient custom functions and <u>virtual table modules</u>. The optional parameter *entrypoint* specifies the initialization function within the extension. If omitted, the entrypoint will default to **sqlite3_extension_init**, which is recommended. See SQLite Wiki's <u>Loadable Extensions</u> article for details and more information.

NOTE. The loadable extension API, as well as the virtual table API, is still beta in SQLite, hence if there are any changes in the API in the future, the changes might also reflect in mSQLite.

sqlite_reload

Reloads the configuration file.

Syntax /sqlite_reload

Remarks

This command allows you to reload the configuration file in case you have made changes but don't want to reload the whole DLL.

sqlite_safe_encode

Encodes data for safe use in files.

Syntax
\$sqlite_safe_encode (data [, delim])

Parameters

data The data to be encoded. *delim* Optional. Delimiter in ASCII used to separate fields.

Return Value

Encoded data on success or **\$null** on error.

Remarks

\$sqlite_safe_encode is used by <u>\$sqlite_fetch_all</u> internally to encode data so it can be safely written into a file.

The characters that are encoded are: $\ (backslash), \ (newline), \ (carriage return), \ (0 (null-byte, in binary data) and whatever delimiter is used, if any. The characters are encoded as an escape sequence <math>\ NN$ where NN is a two-digit hexadecimal number.

In case **\$null** is returned it can mean an error, but it can also happen if you tried to encode an empty string. It should be ok to ignore this, but in case you want to determine whether \$null meant an error or not, you can check the **%sqlite_errno** variable; if it's **\$SQLITE_OK** there was no error.

See Also

sqlite_fetch_all sqlite_safe_decode

sqlite_safe_decode

Decodes safe-encoded data.

Syntax
\$sqlite_safe_decode (data)

Parameters

data The data to be decoded.

Return Value

Decoded data on success or **\$null** on error.

Remarks

\$sqlite_safe_decode is used to decode data encoded by <u>\$sqlite_fetch_all</u> or <u>\$sqlite_safe_encode</u>.

In case **\$null** is returned it can mean an error, but it can also happen if you tried to encode an empty string. It should be ok to ignore this, but in case you want to determine whether \$null meant an error or not, you can check the **%sqlite_errno** variable; if it's **\$SQLITE_ERROR_OK** there was no error.

See Also sqlite fetch all

<u>sqlite_safe_decode</u>

sqlite_autocommit

Turns on or off autocommit mode, or returns its current state.

```
Syntax
$sqlite_autocommit ( conn [, mode ] )
/sqlite_autocommit conn mode
```

Parameters

*conn*The connection identifier.*mode*11 to enable autocommit mode, **0** to disable.

Return Value

If setting the autocommit mode, **1** on success, or **\$null** if there was an error. If getting the autocommit mode, **1** if autocommit mode is enabled, otherwise **0**.

Remarks

When auto-commit mode is enabled every SQL statement is automatically committed after they're executed, unless *\$sqlite_begin* or *BEGIN TRANSACTION* is explicitly used to start a transaction. When disabled, changes to the database are deferred and only committed when *\$sqlite_commit* or *COMMIT TRANSACTION* is used.

Using transactions when doing a batch of updates on database can greatly improve the performance. Disabling auto-commit mode means that you don't have to worry about remembering to start the transaction all the time, all you need to worry about is where you want all the pending changes to be committed.

Auto-commit is enabled by default for new database connections. SQLite doesn't remember the state of auto-commit mode when database is closed, thus you must call this function everytime for a newly opened connection, if you want to disable auto-commit mode by default for that connection.

Enabling auto-commit for a connection will commit all pending changes.

It is usually ok to ignore the return value of \$sqlite_autocommit when used to set autocommit mode, because the only case an error is returned is when an invalid *conn* is specified.

Example

```
; Open a temporary database and disable auto-commit on it
var %db = $sqlite_open()
sqlite_autocommit %db 0
; Create a table and insert a row in it
sqlite_exec %db CREATE TABLE test (text)
sqlite_exec %db INSERT INTO test VALUES ('First row')
sqlite_commit %db
; Insert another row, but this time roll it back
sqlite_exec %db INSERT INTO test VALUES ('Second row')
sqlite_rollback %db
; Print all the rows in the table
var %res = $sqlite_query(%db, SELECT * FROM test), %i = 1
while ($sqlite_fetch_single(%res)) {
  echo -a %i - $v1
  inc %i
}
; Clean up
sqlite_free %res
sqlite_close %db
; Output:
; 1 - First row
; 2 - Second row
```

See Also

sqlite_begin sqlite_commit sqlite_rollback

sqlite_bind_null

Binds null as a parameter for prepared statement.

Syntax

```
$sqlite_bind_null ( statement, param )
/sqlite_bind_null statement param
```

Parameters

statement The prepared statement identifier.

param

The parameter to bind to. Must exist in the prepared query.

Return Value

1 on success, or **\$null** if there was an error.

Remarks

The *param* parameter can either be a numerical index, specified with a **?** in the query, or a named parameter specified with a **:name** in the query. If binding a named parameter, you should also include the colon in the name.

For more information about parameter binding, see <u>Prepared Statements</u>.

See Also

Prepared Statements sqlite_prepare sqlite_bind_field sqlite_bind_param sqlite_bind_value

sqlite_fetch_num

Fetches the current row from a result and then advances to the next row.

Syntax

\$sqlite_fetch_num (result, hash_table)

Parameters

result The result identifier.

hash_table

The name of the hash table to where to store the row data.

Return Value

1 on success; Otherwise **0** if there are no more rows available, or **\$null** if there was an error.

Remarks

\$sqlite_fetch_num is provided for convenience. All it does is call <u>**\$sqlite_fetch_row</u>** with *result_type* set to **\$SQLITE_NUM**</u>

See Also

<u>sqlite_fetch_row</u> <u>sqlite_fetch_assoc</u>

sqlite_fetch_assoc

Fetches the current row from a result and then advances to the next row.

Syntax

\$sqlite_fetch_assoc (result, hash_table)

Parameters

result The result identifier.

hash_table

The name of the hash table to where to store the row data.

Return Value

1 on success; Otherwise **0** if there are no more rows available, or **\$null** if there was an error.

Remarks

\$sqlite_fetch_assoc is provided for convenience. All it does is call **<u>\$sqlite_fetch_row</u>** with *result_type* set to **\$SQLITE_ASSOC**

See Also

<u>sqlite_fetch_row</u> <u>sqlite_fetch_num</u>