# SQLitening

**Client/Server implementation**

of the

**SQLite Database.System**

Version 1.30

## SQLitening

**SQLitening Database System**

SQLitening is a client/server implementation of the very popular SQLite database.

SQLitening is a programmer's library in standard Win32 DLL form. It is installed as a standard Windows Service. In addition to client/server mode, the library allows the programmer to also access SQLite databases in local mode. In either mode (local or client/server), the database is extremely fast and robust.

Installation is a breeze - you simply copy a couple of DLL's to the folder where your application resides. If you work in client/server mode, you create a folder on your server and start the SQLitening Windows Service from the Administration program. You may need to modify the standard configuration text file to set permissions and port numbers/host names. Simple.

[Introduction](Introduction)

[Developer Guide](Developer Guide)

[User Guide](User Guide)

[Misc](Misc)

# Introduction

[Release History](#)

[Project Files](#)

[ReadMe](#)

# Release History

======<[ Version 1.3  November 1, 2009]>======

**1.** Removed the call to sqlite3_thread_cleanup in SQLiteningServer.Bas. It had become obsolete.

**2.** Added calls to sqlite3_initialize at server start and sqlite3_shutdown at server stop in SQLiteningServer.Bas.  These are new SQLite functions.

**3.** Added call to sqlite3_shutdown in SQLitening.Bas when process detaches.

**4.** Change SQLitening.Dll so all sets are automatically closed when slClose is called.

**5.** Change SQLiteningServer.Exe to timeout a connection after a number of minutes of no activity.  The default number is 30.  You can change it  in the config file.  You can do an empty string slExe if you need to keep a connection active.  Error %SQLitening_SendOrReceiveError will occur if you attempt to use a connection that has timed out.

**6.** Added the SystemTemp database to SQLiteningServer.Exe to be able to store stuff like the named locks and connection data.

**7.** Fixed a bug in SQLiteningServer.Exe that was sending back the wrong return value if the slSel was huge.

**8**  Changed the error handling on the following ruts to be consistent with other ruts that returned string.
> slGetColumnName
> slGetTableColumnNames
> slGetTableNames

**9.** Added rut slSetRelNamedLocks which will set or release named lock(s). This can be used to implement application locks or to simulate record  locks.  Names are locked on a server-wide basis.  If a name has been locked by one client, the server blocks any request by another client for a lock with the same name.  This allows clients that agree on a given lock name to use the name to perform cooperative advisory locking.  But be aware that it also allows a client that is not among the set of cooperating clients to lock a name and thus prevent any of the cooperating clients from locking that name.  One way to reduce the likelihood of this is to use lock names that are database-specific or application-specific. Named locks are only used in remote mode.  They are ignored when running in local mode.  Will also optionally do a Sel command but only if the lock request was successful.

**10.** Added the following ModChars to slSel, slSelAry, and to the SelStatemnt of slSetRelNameLocks:

> **Bn** = Do a Begin Transaction before doing the Sel command. The type of Begin is controlled by the value of n as follows:
>> 0 = Deferred. This is the default if n is omitted.
>> 1 = Immediate.
>> 2 = Exclusive.
>
> This allows for database locking and selecting in one trip to the server.
> **R** = Release all named locks owned by this connection.

**11.** Added the following ModChars to slExe:

> **R** = Release all named locks owned by this connection.

**12.** Added rut slGetStatus which returns the requested status which is normally a delimited list of returning data. The following requests are currently valid:

> 1 = Return all named locks.
> 2 = Return all connections.
> 3 = Return SQLitening flag settings.

**13.** Added the server name suffix to SQLiteningServerAdmin.Bas messages and changed the log to also show the suffix. Having a unique suffix allows multiple services (servers) to be running on same computer.

**14.** Added a new dialog to SQLiteningServerAdmin which allows for the following tools:

> Refresh the Config Flags and FACT
> List all Active Connectons
> Kill one Active Connection

You already had the ability to "Refresh" but the other two are new.

**15.** Added two new items to the config file.

> **MaxChunkSize = Number K**
> Controls the size of Row Data Chunks which are returned from Select statements. The value is in K so the actual size is * 1000. Default is 500.
>
> **ConnectionTimeOut = Number Minutes**
> Control the number of minutes the server will wait to receive a message from an active connection. Default is 30.

**======<[ Version 1.20 June 19, 2009 ]>======**

**1.** Changed SQLitening.Dll to close the database when detaching and running local. This will process and remove any pending journal file. This does not change the end result it just cleans up the journal file sooner.

**2.** Fixed Load/Unload problem with RunProc.

**3.** Added the slIsOpen function which will return %True if a database is open.

**4.** Added slPushSet and slPopSet.  Push will save the data about an open set while pop will restore the data.  This allows you to reuse a set number with out first closing it.

**5.** Added a global flag (AreConnected) to improve the control of running remote or local and allowing for retrying to connect to server if first attempt(s) failed.

**6.** Fixed bug in server that would cause a crash if Close was called at wrong time.

**7.** Added bind blob and bind text examples to SQLiteningProcA.Bas.

**8.** Added another conditional compile Server Exit #4 which will fire for ever slOpen.  This is the place to create SQLite custom function.

**9.** Added Exit #4 to SQLiteningServerExits.Bas and included as example of two SQLite customer functions (MyF0 and MyF2).

**10.** Added the "S" ModChar the both slGetFile and slPutFile.
S  = Open file with Shared lock, default is both Read and Write locks.
This only applies to standard files, not Ini.

**11.** There will now be three supported API's. Basic, Special, and Universal.
   See doc in SQLitening.Txt.  Support for SQLiteningVB is discontinued.

**12.** Included the zip file UserMods.Zip which contains user contributed additions and/or modifications to SQLitening soruce code.

**13.** Update SQLite3.Dll to version 3.6.15 dated 06-15-2009.


**=======<[ Version 1.11 January 17, 2009 ]>======**

**1.** Changed slGetInsertID to prevent random GPF's.   My testing says that PB9 will handle Quads returning from C but only if you use early binding (no call Dword).  SQLitening.Dll must use late binding cause it will dynamically call either SQLite3.Dll or SQLiteningClient.Dll therefore PB9 doesn't help here. SQLiteningServer, On the other hand, does use early binding so PB9 can do it's thing there. Local mode will no longer call the SQLite API to obtain the last inserted ID instead it will get it using the last_insert_rowid() function.  This is a little slower than the API call but should eliminate the GPF problem. Remote

mode will continue calling the API but have removed the assembler stuff and will let PB9 do it's Quad thing.

**2.** Changed a failed connect to remain in remote mode so all following commands will also fail.  It was returning to nurtural mode so following commands would process in local mode --- not good.

**3.** Added a new optional parameter to the slConnect command called wsOutData.

- OutData, if parm is passed and if return is zero, will contain the    following values delimited by the $BS character. There is only one value being returned now but others may be added in future so use the parse$ command.
    - 1 = The unique TcpFileNumber assigned by the server to this connection.  This can be used whenever a unique number for a connection is needed.  This same value is passed to ceratin exits.

**IMPORTANT!!!**  All programs using slConnect should be recompiled when using this release.

**4.** Added another parm to be passed to a user coded proc.  This new parm (byval rlTcpFileNumber as Long) will contain the unique number assigned by the server to a connection.

**IMPORTANT!!!**  All user procs should be recompiled when using this release.

**5.** Changed the starting of SQLiteningServerMonitor to be a conditional compile.  If SQLiteningServer is compiled with %CompileStartServerMonitor = %True then SQLiteningServerMonitor will be started when SQLiteningServer is started. %CompileStartServerMonitor = %False is the default.

**6.** Added the concept of 'Exits' to SQLiteningServer.  The term 'Exit' comes from the main process (SQLiteningServer.Exe) 'Exiting' to a sub process (SQLiteningServerExits.Dll) which contains your custom code and then waiting for the return code.  If the return is zero then the server will continue to process else will return that error value to the calling application program.  Exits are placed at strategic points in the process loop of SQLiteningServer.  More exits may be added in future. This sub process must be thread safe, it can access SQLite3.Dll directly but it can not use SQLitening.Dll because it is not thread safe.  See the sample source SQLiteningServerExits.Bas for what exits are available and what parameters are passed and other information.  There currently are 5 exits available and are controlled by the following conditional compile equates:
- %CompileExit_1_Connect,
- %CompileExit_2_Disconnect,
- %CompileExit_3_Access

- %CompileExit_101_Start,
- %CompileExit_102_Stop

The default is that they are all %False. Because the exits are activated only if you want them, there is no overhead for the ones who don't use them.

**7.** Added the 'Q' and 'c' ModChars to slSelAry. The 'Q' will return a one dimension array with delimited column values.  The 'c' will not return the column names in entry zero.
- Q# = Return a one dimension array where each column is delimited by a single character.  That character is determined by the ascii value at #. The default is to return a two dimension array.
- c = Do not return the column names as a row of data at index zero.  The default is to return the column names as row zero.

**8.** Change SQLitening.Dll so all sets are automatically closed when the Dll is detached.

**9.** Updated SQLite3.Dll to version 3.6.7.


**=======<[ Version 1.1 November 6 2008 ]>=======**

**1.** Removed the "P" ModChar from slFX, slFNX, and slSelAry. Just use "D" and "U" ModChars to Decrypt and Uncompress.
Purpose: The "P" ModChar was redundant and bloated the code.

**2**. Deleted the slGetFieldType routine and replaced it with slIsFieldNull. The slGetFieldType could not be handled the same way in both local and remote mode. It was decided that knowing the data type other than Null was not needed.
Purpose: Make local and remote mode are the same.

**3.** Added the slGetHandle routine. This will return either the open database or open set handle.
Purpose: To be able to call SQLite directly. Only allowed in local mode.

**4.** Added the %ReturnAllErrors conditional compile equate.
Purpose: So SQLitening.Dll can be compiled with Return Errors as the default.

**5.** Added the slConvertDat function which accepts a string and returns a converted string. The posible conversion are:
Compress, Encrypt, Uncompress, Decrypt, Hexadecimal
Purpose: This returned hex string can be used in a where clause for Blob data. It can also be used to insert or update Blob data (slExeBind is more efficient).

**6.** Added the ability to have a user coded monitor process running along with the

server process. SQLiteningServer will now shell
to the SQLiteningServerMonitor.Exe process when it starts. This process will run
as long as the server is running, it will exit when server is stopped. If this monitor
process is not required then just delete SQLiteningServerMonitor.Exe.
Purpose: This is the place you can code things like writing back up copies or do
any other database or file maintenance using all the power of SQLitening in local
mode.

**7.** Added the ability to have user coded procs within a Dll called
SQLiteningProcs.Dll. These procs will allow you to write your own code which will
run at the server accessing SQLite in local mode. You can call a proc using
slCallProc or the SQLite function load_extension (no parms allowed using
load_extension). These procs talk to SQLite directly without going thru
SQLitening. Not using SQLitening is required cause SQLitening.Dll is not thread
safe and the server starts a new thread for each connection. These procs will
also work fine in local mode. This concept is similar to database procedures in
Oracle and MySQL except here you get to code in Basic rather that their
proprietary language.Purpose: Using procs can greatly improve response time by
reducing messages to/from server.

**8.** Added the ServiceNameSuffix entry to the server config file.Purpose: To allow
multiple services (servers) to be running on same
computer. You could, for example, have both a Test and Prod server.

**9.** Added SQLiteningVB.Dll which is a bridge DLL for Visual Basic to handle the
ByVal string problem. The SQLiteningVB.Ini file has
also changed. A VB program will call SQLiteningVB which in turn will call
SQLitening for the ruts that use ByVal strings. These
are not fully tested because I do not have VB.
Purpose: To allow Visual Basic to use SQLitening.


**======<[ Version 1.06 October 24, 2008 ]>======**

**1.** Changed version numbers scheme. You can always tell what version a file is
by looking and it's time. The hour will be the major
version number while the minute will be the minor number.

**2.** Made changes to the include file so all programs must be recompiled.Also be
sure you install all Dll's and Exe's as a set.

**3.** Changed slExeBind to accept a string of BindDat entries instead of a string
array. A BindDat entry is built using a new routine called
slBuildBindDat. This will make slExeBind easer to code, run faster, and easer to
understand. See new doc for slExeBindDat and check
out slBuildBindDat.

IMPORTANT: All programs that used slExeBind must be changed and recompiled.

**4.** Completely recoded error processing. If you are returning errors then the slGetError or a new routine called slGetErrorNumber will
return the last error. slGetError returns a string containing "number = description" while slGetErrorNumber returns a long
containing only the number. Since the last error number is stored in a global long, slGetErrorNumber is very fast where slGetError
must do a call to SQLite.

**5.** The "E" Modchar now has an optional 'm' modifier which is the message display modifier and can be:
    0 = No message is displayed. This is the default.
    1 = Display a warning message with OK button. Error is returned when OK
       pressed.
    2 = Display a question message with OK and Cancel buttons. If they press
       OK, error is returned. If they press Cancel, will exit process.
So now if you do slSetProcessMods "E1" then if any error occurs SQLitening will display the error message and then return to your
program.
**6.** Changed SQLiteningClient.Dll to return all errors rather than exiting processing. Added following error number. -18 = Error sending or receiving message" ' Server went away or message length error

**7.** Added "E" ModChar to slGetRow. If an error occurs and you are returning errors then will return %False.

**8.** Added "D" ModChar to slSel which will allow duplicate column names.Not recommended if using slFN or slFNX because you will always get the first value returned. SQLite does not normally return qualified column names. SQLite will return C1 twice if you Select T1.C1, T2.C1. So the solution is to alias one of them with the As clause as follows:
Select T1.C1, T2.C1 as C1Again. There is a Pragma called "full_column_names" which forces SQLite to return qualified names, but does not seem to work if you Select *. Read up on it and use if you like. I like using an alias because it is less code.

**9.** The change to slExeBind allows for VB and VB.Net to use SQLitening as is. No special shell needed. So I added two include files.
SQLiteningVB.Inc and SQLiteningVBNET.Inc. Did light testing with VB5 and VB 2008 (.Net). Seems to work fine. slSelAry is removed from the include files because array passing is not compatibly.slSelAry is not required -- just handy sometimes. Also the return value for slGetInsertID is changed to Currency for VB.

**10.** SQLitening.Txt has many changes.


**=======<[ Version 5 -- September 10, 2008 ]>=======**

**1.** Fixed bug in checking for duplicate column names in Select statement.

**2.** Changed slClose and slCloseSet to never raise an error.

**3.** Added the "E" ModChar to the two extended get field commands (slFX and slFNx).
E = Return errors. The return value will be an empty string. The error number, preceded by #, will be set back into ModChars.
The non extended get field commands (slF and slFN) will still exit process if there is an error.

**4.** Added global flags to slSetProcessMods that will
return all errors. The two new ModChars are:
E = Return all errors. SQLitening will not exit the process except
for certain unusually situations like when it can not load a
library. This setting is global.
e = Only return errors if the "E" ModChar is passed with the command.
This is the default. If no "E" passed then errors will cause a
message to display and the process to exit. This setting is global.

**5.** Added following ModChars to slConnect for security reasons:
   u = Do not pass user name.
   c = Do not pass computer name.
   i = Do not pass IP address.

**6.** Eliminated the User entry in log. The user data is now placed in the Conn entry. That eliminated the LogUser entry in the config file.

**7.** Changed when the Conn entry is logged and add user data to it.Added the LogInvalidInMessage entry to the config file.
LogInvalidInMessage = Yes or No --- Controls the logging of invalid incoming messages. If Yes and the message is invalid (first is not slConnect or is a wrong length) is received then an Eror message is logged. If omitted then will default to No. No will prevent hackers/attackers from filling the log.

**8.** Logging of Conn/Dcon will now only occur if the first message is a valid slConnect. Hackers/attackers won't fill log.

**9.** Changed slConnect so that if already connected the will just return rather than establish another connection.

**=======<[ Version 4 -- August 26, 2008 ]>======**

**1.** Fixed documentation for slAttach, slOpen, slGetFile, and slPutFile

**2.** Fixed security bug that allowed slOpen to create a file without correct password.

**3.** Fixed bug in slBuildInsertOrUpdate.

**4.** Changed slOpen and slAttach to return error -9 (File does not exist) rather than error 14 (Unable to open the database file) if the file does not exist.

**5.** Changed slConnect if $NUL is passed as the Server then will run in local mode.

**6.** Add ability to force a value to be numeric in slBuildInsertOrUpdate.

**7.** Fixed bug in slExe error handling.

**8.** Added checking for duplicate column names in Select statement. Will raise error -13 if occurs. SQLite allows duplicate column names in the Select statement which would cause slFN and slFNX to return undesirable stuff.

**9.** Strengthened the documentation in SQLitening.Bas and SQLitening.Txt.

**10.** Added a new function called slSelAry which will return a two dimension array containing the column data from all rows. This will allow you to navigate a record set both forward, backward, and directly. Sample usage is in ExampleC.Bas.

**11.** Changed the password checking to include read-only access. A password containing the percent sign (%) character will require the file to be opened as read-only.

**12.** Fixed bug in slExeBind.

**13.** Fixed bug in slGetColumnCount.

**14.** Added a ReadMe.Txt file in Doc folder.


**======<[ Version 3 -- July 15, 2008 ]>======**

**1.** Changed slOpen to use the new sqlite3_open_v2 API which was added in version 3.5.0 of SQLite3. This new open allows for read only and temporary

database on disk. It also creates a new database only if requested. The temporary database on disk could be very useful (still can have the :memory: one). Removed the ModChar S. It was no loger needed with the new sqlite3_open_v2 API.

**2.** Eliminated slIsSetEmpty and slIsRowThere. These were not clean routines and their use could have caused confusion. Using "if slGetRow" will accomplish same results and is much cleaner.

**3.** Added ModChar of "C" to slGetRow which will close the set even if there is another row available. This has use when all you want
to do is check if an set has rows.
slSel "Select 1 from T1 where Key='ABC'"
if slGetRow(0, "C") then
' has rows action
else
' has no rows action
end if

**4.** Changed slIsSetOpen to slIsSetNumberValid to be consistent.

**5.** Changed slGetChangeCout to use ModChar instead of Flag to be consistent.

**4.** Added slIsColumnNumberValid and slIsColumnNameValid. These along with slIsSetNumberValid should prevent aborts and allow for more programmer control.

**5.** Improved performance when checking for valid column number.

**6.** Added column number, column name, and set number audits to appropriate routines to prevent GPFs.

**7.** Error number -13 (Invalid column name or number) was added.

**8.** Changed SQLitening.Txt to be in alpha sequence rather than by group.


**========<[ Version 2 -- July 7, 2008 ]>======**

**1.** Eliminated error -13 which was used for Invalid parms. The current code can no longer rais this error.

**2.** Eliminated the slEscapeSpecialChars routine. It did only one PB replace command which replaced all ' with ''.

**3.** Added slIsSetOpen routine. This was a forum request.

**4.** Enhanced slExeBind so it can bind multiply fields in one call. It can also encrypt and/or compress a field.

**5.** Enhanced slFX and slFNX to be able to get fields and decrypt and/or uncompress.

**6.** Added the K ModChar to slSetProcessMods which will set/unset the cryption key.

**7.** There is a new Dll called SQLiteningAuxRuts.Dll. This is were the Rijndael cryption code resides as well as the calls to Zlib for compression. I added a Dll so there would be no additional memory usage for those that do not use cryption nor compression. This aux dll resides only with the client -- server is not concerned. Note that the encryption/compression occurs at the field level. Not the row nor column level. So if you encrypt/compress with slExeBind then you must also decrypt/uncompress with either slFX or slFNX. I used Rijndael cause it is much faster and smaller then BlowFish and was selected by the NIST as Advanced Encryption Standard. Whatever that means but it impressed me.

**8.** Added two new options to the General section of the config file:
    CreateDatabaseAllowed = Yes or No --- Controls the creation of new
        databases. If Yes then clients are allowed to create new databases on the
        server. If omitted then will default to No
    TrimLogManually = Yes or No --- Controls the trimming of the log when it
        becomes larg e (> 600K). If Yes then no automatic trimming will occur. If
        no then will automatically trim 100K from front of log when it becomes
        large. If omitted then will default to No
**9.** Enhanced the Adim refresh so it will now refresh the config flags as well as the FACT

**10.** Included a second simple example program which does encryption and compression.

**11.** Enhanced almost all of the routine comments. The updated comments are changed in both SQLitening.Bas and SQLitening.Txt. This area should settle down in future.


**========<[ Version 1 -- June 29, 2008 ]>=======**

# Project Files

## ==============<[ Files in the Bin Folder ]>==============

**ExampleA.Exe** - Sample program.
**ExampleB.Exe** - Sample program.
**ExampleC.Exe** - Sample program.
**sample.db3** - Database used by some of the example programs.
**SQLite3.Dll** -The database engine from SQLite.  If running local then it must be accessible on the client computer.  If running remote then it must be accessible on the server computer.
**SQLitening.Dll** - Contains the routines called by your application program.  It must be accessible on the client computer.
**SQLiteningAuxRuts.Dll** - Contains the compression and encryption routines called by SQLitening.Dll.  It muse be accessible on the client computer but only if your are compressing or encrypting.
**SQLiteningClient.Dll** - Contains the routines called by SQLitening.Dll to communicate with the server.  It must be accessible on the client computer but only if you are running in remote mode.  Not used in local mode.
**SQLiteningS.Dll** - Contains routines, called from any program that supports OLE string parameter passing but only ByRef (like Visual Basic).
**SQLiteningServer.Cfg** - The file used to configure the server.  See [Documentation of Sections and Entities] in file.
**SQLiteningServer.Exe** - The program that runs on the server as a service.
**SQLiteningServerAdmin.Exe** - The program that runs on the server.  Is used to install, start, stop, and uninstall the service.
**SQLiteningU.Dll** - Contains routines, that can be called by almost any program, to access a SQLite databases and flat files residing locally or remotely (using SQLitening Server). This can support any language that can call a standard Dll.
**Zlib.Dll** - Contains the compression routines called by SQLiteningAuxRuts.Dll, SQLiteningClient.Dll, and SQLiteningServer.Exe.  If running remote or doing compression in local mode then it must be accessible on the client computer.  If running remote then it must be accessible on the server computer.

## ==============<[ Files in the Doc Folder ]>==============

**ReadMe.Txt** - Is this file.
**ReleaseHistory.Txt** - Contains history of changes in each released version.
**SQLitening.Chm** - Help file built by Rolf Brandit.
**SQLitening.Pdf** - Help file built by Rolf Brandit.
**SQLitening.Txt** - Contains documentation of each routine in SQLitening.Dll.

## =============<[ Files in the Src Folder ]>==============

**SQLitening.Bas** - Source.
**SQLitening.Ico** - Icon.
**SQLitening.Pbr** - Resource object.
**SQLitening.Rc** - Resource source.
**SQLiteningAuxRuts.Bas** -Source.
**SQLiteningClient.Bas** - Source.
**SQLiteningProcsA.Bas** - Soruce sample.
**SQLiteningS.Bas** - Source.
**SQLiteningServer.Bas** - Source.
**SQLiteningServerAdmin.Bas** - Source.
**SQLiteningServerExits.Bas** - Source sample.
**SQLiteningServerMoniter.Bas** - Source sample.
**SQLiteningU.Bas** - Source.
**UserMods.Zip** - Contains user contributed additions/modifications.

## ============<[ Files in the Inc Folder ]>==============

**SQLitening.Inc** - Include declares for the Basic API.
**SQLiteningS.Inc** - Include declares for the Special API.
**SQLiteningU.Inc** - Include declares for the Universal API.

## ============<[ Files in the Examples Folder ]>==========

**ExampleA.Bas** - Source
**ExampleB.Bas** - Source
**ExampleC.Bas** - Source

## ============<[ Notes ]>===========================

1. PB9 is required if you want to compile any SQLitening program.

## ReadMe

**PUBLIC DOMAIN SOFTWARE**

The author or authors of this code dedicate any and all copyright interest in this code to the public domain. Anyone is free to copy, modify, publish, use, compile, sell, or distribute the original code, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

Fred Meier - November 2009

# Developer Guide

[SQLitening Routines](#)

[WhitePapers](#)

# SQLitening Routines

*Many of these Functions provide very simple **Code Examples** to show its usage. Most examples are based on the "**sample.db3**" database that is part of the package. Example code will be further extended. For more advanced usage consider ExampleA.bas, ExampleB.bas, and ExampleC.bas in the Examples folder.*

slAttach
slBuildBindDat
slBuildInsertOrUpdate
slClose
slCloseSet
slConnect
slConvertDat
slDisconnect
slExe
slExeBind
slF
slFN
slFX
slFNX
slGetChangeCount
slGetColumnCount
slGetColumnName
slGetColumnNumber
slGetError
slGetErrorNumber
slGetFile
slGetHandle
slGetInsertID
slGetRow
slGetStatus
slGetTableColumnNames
slGetTableNames
slGetUnusedSetNumber
slIsColumnNameValid
slIsColumnNumberValid
slIsFieldNull
slIsOpen
slIsSetNumbervalid
slOpen
slPopDatabase
slPopSet
slPushDatabase
slPushSet

## slAttach

### slAttach *(rsFileName String, rsAsName String, [rsModChars String]) Long*

Will attach the passed FileName using the AsDatabaseName. FileName, may also contain an optional password. This password is separated from the file name by the $BS character.

*ModChars:*
- Em = Return errors. This will override the global return errors flag.
- m is the optional message display modifier and can be:
  - 0 = No message is displayed. This is the default.
  - 1 = Display a warning message with OK button. Error is returned when OK pressed.
  - 2 = Display a question message with OK and Cancel buttons.
  - If they press OK, error is returned. If they press Cancel, will exit process.
- e = Do not return errors, display message and exit process. This will override the global return errors flag.
  - Returns zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

*--- Local Mode ---*
If the FileName is not fully pathed then it is assumed to be relative to the current folder which is the same folder, unless changed by chdir, that the first .Exe was run from. If fully pathed then no assumptions are made. Since this running on a local computer SQLitening allows the files to be located anywhere on the local hard drives or local network drives.

*Examples:*
If your .Exe started in C:\Apps\MyApp then:
If FileName is X\Y\Able.Sld then will assume file is in C:\Apps\MyApp\X\Y.
If FileName is ..\Y\Able.Sld then will assume file is in C:\Apps\Y.
If FileName is C:\Able.Sld then no assumption.

*--- Remote Mode ---*
The FileName is assumed to be relative to the folder which the service is running from. Since this is running on a remote server SQLitening can not allow the user to access files anyplace on the server. Access is denied to any FileName that has a colon, a double dot, or begins with a backslash. This will insure that the file is in same folder as the service or below it.

*Examples:*
If your service is running from C:\SQLitening then:

If FileName is Data\Able.Sld then will assume file is in C:\SQLitening\Data.
If FileName is ..\Y\Able.Sld then will get error -8 Access Denied.
If FileName is C:\Able.Sld then will get error -8 Access Denied.


**Code Example:**

```
' Create a new database
slOpen "DBFirst.db3", "C"
slExe "Create Table If Not Exists Table1 (Field1, Field2, Field3)"
slClose

' Create a second database and attach the first
slOpen "DBSecond.db3", "C"
slExe "Create Table If Not Exists Table1 (Field1, Field2, Field3)"
slAttach "ExampleB_New1.Sld", "One"
```

## slBuildBindDat

## slBuildBindDat *(rsData String, [rsModChars String]) String*

Returns a BindDat entry which is a specially formatted string required by slExeBind. Data contains the value you want converted into a BindDat.A BindDat(s) is required to be passed to slExeBind. The returned data may also be compressed and/or encrypted. If an error occurs then the return value will be an empty string. Use slGetError or slGetErrorNumber to determine the error.

**ModChars:**
- B = Bind as Blob. This is the default.
- C = Compress the data.
- N = Encrypt the data.
- T = Bind as Text. Default is to bind as Blob.

*Note:*
If both C and N then will first compress and then encrypt.

**slBuildInsertOrUpdate**

## slBuildInsertOrUpdate *(rsTable String, rsValues String, [rsColumns String, rsWhere String]) String*

Returns either an Insert or Update SQL statement which is ready to pass to slExe or slExeBind. Many times you want to update a record if its there or add it if it is not there. The SQL syntax makes this clumsy, this routine can help. Values contains a delimited text string which is $NUL separated. $NUL was used because SQLite does not allow it to be in any of the values. Values are passed as alpha or non alpha (numeric, NULL, or ?). A value is considered numeric if it contains only +-.0123456789 or ends with $VT. If it does not end with $VT then there can be only one + or - and it must be in the first position. There can be only one decimal point. If it ends with $VT (used for passing expressions like 126 * 85) then the $VT will be removed and the remaining will be considered numeric. Alpha values will have all embedded single quotes changed to two single quotes and they will be enclosed within single quotes. If an alpha value is already enclosed in single quotes then it will NOT be modified.

*Examples:*
-123.45 -----> -123.45
-----> '' (empty value passed)
? -----> ?
'?' -----> '?'
+45- -----> '+45-'
126 * 85 -----> 126 * 85 (the is $VT)
AB'CD -----> 'AB''CD
Null -----> Null
'Null' -----> 'Null'
'AJ'XY' -----> 'AJ'XY' (SQLite will raise error)

If Where is omitted or empty then will build an Insert statement else will build an Update statement appending " Where " and the Where value. Columns contains a comma delimited list of the column names. Column names are optional for Insert but required for Update. This routine does not call SQLite. No error can occur. Normal usage is as follows:

slExe slBuildInsertOrUpdate("TableA", "ABC" & $NUL & "123")
slExe slBuildInsertOrUpdate("TableA", "ABC" & $NUL & "123", "ColA,ColB", "rowid=1")

**Code Examples:**
These are two small examples to save a new record and an existing modified record.

```
'Create a new record
Function SaveNewRec()As Long
        Local fstr      As String       'the fields string
        Local vstr      As String       'the values string

        fStr = "MANUF,REDREF,PRODUCT,TYPE"
        vStr = "MyCompany" & $NUL & "'003214523'" & $NUL & _
               "MyProduct" & $NUL & "MyType"

        slExe slBuildInsertOrUpdate ("Parts", vstr, fStr)
End Function

'Update an existing record
Function UpdateRec()As Long
        Local fstr      As String       'the fields string
        Local vstr      As String       'the values string
        Local wstr      As String       'the where string

        fStr = "MANUF,REDREF,PRODUCT,TYPE"
        vStr = "MyCompany" & $NUL & "'003214523'" & $NUL & _
               "MyProduct" & $NUL & "MyType"
        wStr = "rowid = 123"

        slExe slBuildInsertOrUpdate ("Parts", vstr, fStr, wStr)
End Function
```

## slClose

## slClose

Closes the current database handle and sets it to nothing.  Also will close all sets.  This is optional.  All databases and sets will be automatically closed when your program ends. If you are done with a database but your program will continue to run for some time then it is a good idea to close it.  Will not raise error if database is already closed.  Normally no error will occur.  You can use slGetError or slGetErrorNumber to be sure.

**Code Example**:

**slClose**

## slCloseSet

## slCloseSet *[rlSetNumber Long]*

Finalize the set handle passed in SetNumber and sets it to nothing. A set is automatically closed when you NextRow through all the rows or when your program ends. You need this function only when do not NextRow through all the rows and you need the set closed because you want to reuse the SetNumber. Will not raise error if set is already closed.

**Code Example**:

```
' Process records of Set 1
Do While slGetRow(1)
        ' check the value of field 3
        if slF(3, 1) >= "AB100" then Exit Do
Loop
slCloseSet(1)
```

**slConnect**

## slConnect *([rsServer String, rlPort Long, rsModChars String, wsOutData String]) Long*

Will connect to a SQLitening server.  This is only required if you want to run in remote mode. Do not call this routine or set Server to $NUL if you want to run in local mode. If Server is omitted or empty it will default to LocalHost. Server can be a host name, or an IP address (n.n.n.n)   If Port is zero or omitted it will default to 51234.  If already connected then will just return OK, no error is raised.

**ModChars:**

- **Em** = Return errors. This will override the global return errors flag.
    m is the optional message display modifier and can be:
        0 = No message is displayed.  This is the default.
        1 = Display a warning message with OK button.  Error is returned when OK pressed.
        2 = Display a question message with OK and Cancel buttons.
            If they press OK, error is returned.  If they press
            Cancel, will exit process.
- **e**  = Do not return errors, display message and exit process. This will override the global return errors flag.
- **Tn** = Where n is the how long a TCP SEND/ RECV should wait for completion, in milliseconds (mSec). The default timeout is 30000 milliseconds (30 seconds).  This wait should be set to be at least 3 times as long as the wait for a database lock which that default is 10000 milliseconds (10 seconds) and is set in slOpen and can be changed in slSetProcessMods.
- **u**  = Do not pass user name.
- **c**  = Do not pass computer name.
- **i**  = Do not pass IP address.

**OutData**, if parm is passed and if return is zero, will contain the following values delimited by the $BS character.  There is only one value being returned now but others may be added in future so use the parse$ command.
        1 = The unique TcpFileNumber assigned by the server to this connection. This can be used whenever a unique number for a connection is needed. This same value is passed to ceratin exits.
Returns zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

**Code Example:**
'IP could be a local or global string variable holding the IP adress of the server,
'i. e. "192.168.1.42" in a LAN, "83.122.201.123" or "myserver.no-ip.org"

```
'for an Internet based connection (WAN),

FUNCTION DBConnect()as long
        Local IErr as String

        slConnect IP, 0, "E0"
        'Error handling
        IErr = slGetError
        IF VAL(IErr) <> 0 THEN
                MSGBOX IErr,  0, "Connection Error"
        END IF
END FUNCTION
```

## slConvertDat

## slConvertDat

Returns a string which has been converted per ModChars. Data contains the value you want converted. If an error occurs then the return value will be an empty string. Use slGetError or slGetErrorNumber to determine the error.

### *ModChars:*
- C = Compress
- N = Encrypt
- D = Decrypt
- U = Uncompress
- X = Hexadecimal

### *Note:*
If C or N then D and U are ignored.
If both C and N then will first compress and then encrypt.
If both D and U then will first decrypt and then uncompress.

### CAUTION:
Requesting to uncompress a field that is not compressed will cause unpredicable results!

## slDisconnect

Disconnects from the server. This is optional. A connection will be automatically disconnected when your program ends. If you are done with a connection but your program will continue to run for some time then it is a good idea to disconnect. Also needed if running remote and you now want to start running only local. No error will occur.

# slExe

## slExe *(rsStatement String, [rsModChars String]) Long*

Executes one or more statements. Each statement is separated by a semicolon.

**ModChars:**
- **C** = Must change 1 or more rows. This will only work for the last statement.
- **Em** = Return errors. This will override the global return errors flag.
  m is the optional message display modifier and can be:
  - **0** = No message is displayed. This is the default.
  - **1** = Display a warning message with OK button. Error is returned when OK pressed.
  - **2** = Display a question message with OK and Cancel buttons. If they press OK, error is returned. If they press Cancel, will exit process.
- **e** = Do not return errors, display message and exit process. This will override the global return errors flag.
- **R** = Release all named locks owned by this connection after doing the Exe.

**CAUTION:** If ANY of the statements returns Busy then will restart with the first statement. Use Begin Immediate to prevent this or set ProcessMods to %gbfDoNotRetryIfBusy.

Returns zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

**Code Examples:**

```
'1. Create a new table in a database
slExe "Create Table If Not Exists Table1 (Field1, Field2, Field3)"
```

```
'2. Create a new table in a database in a transaction (for more reliability)
Function CreateTable()As Long
      ' Begin transaction
      slExe "Begin"
      slExe "Create Table Customers (Name, Adress, City,Zip, Phone)"
      'Create index(es)
      'Create triggers
      'etc
      ' End transaction
```

```
        slExe "End"
End Function


'How to create an empty database see slOpen.


'3. Update a record
Function UpdateRec()As Long
        Local fstr      As String      'the fields string
        Local vstr      As String      'the values string
        Local wstr      As String      'the where string
        Local lErr      As String

        fStr = "MANUF,REDREF,PRODUCT,TYPE"
        vStr = "MyCompany" & $NUL & "'003214523'" & $NUL & _
                "MyProduct" & $NUL & "MyType"
        wStr = "rowid = 123"

        slExe slBuildInsertOrUpdate ("Parts", vstr, fStr, wStr)
End Function
```

## slExeBind

## slExeBind *(rsStatement String, rsBindDats String, [rsModChars String]) Long*

Statement contains one (no multiples like slExe) Insert or Update statement.  Will replace all of the '?' expressions in Statement with the corresponding BindDat entry passed in BindDats.  A BindDat entry is the specially formated string returned by the slBuildBindDat function.  Pass BindDats as multiply concatenated BindDat entries to handle multiple '?' expressions.  The first '?' will be replaced with the first BindDat entry, the second '?' with the second BindDat entry, etc. Excess '?' expressions will be set to Null while excess BindDat entries will raise an error.

**The following example will insert three columns as follows:**
ColName1 = Blob, ColName2 = Compressed Text, ColName3 = Compressed Encrypted Blob.

slExeBind slBuildInsertOrUpdate("TableName", _
        "?" & $NUL & "?" & $NUL & "?" & $NUL, _
        "ColName1, ColName2, ColName3"), _
        slBuildBindDat("This is some Blob data") & _
        slBuildBindDat("This is some compressed Text", "TC") & _
        slBuildBindDat("This is a compressed and encrypted Blob", "CE")


*ModChars:*
- **Em** = Return errors. This will override the global return errors flag.
      m is the optional message display modifier and can be:
            0 = No message is displayed.  This is the default.
            1 = Display a warning message with OK button.  Error is returned when OK pressed.
            2 = Display a question message with OK and Cancel buttons.
            If they press OK, error is returned.  If they press Cancel, will exit process.
- **e**  = Do not return errors, display message and exit process. This will override the global return errors flag.
- **R** =  Release all named locks owned by this connection after doing the ExeBind.

Returns zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

**sIF**

**sIF** *(rlColumnNumber Long, [rlSetNumber Long]) String*


Returns the value for the field passed in ColumnNumber from the current row for the set number passed in SetNumber. Due to the frequent usage of this routine it has a very short non-descript name. If an invalid ColumnNumber or SetNumber is passed then will exit process.


**Code Example**:

```
Local NumFlds      as String
Local Rec          as String
Local i            as Long

' Process records
Do While slGetRow
        ' When a row is returned, you use the sIF, sIFN, sIFX, sIFNX functions.
        Rec = ""
        For i = 1 To slGetColumnCount
                Rec = Rec & sIF(i) & " - "
        Next
        Listbox Add hDlg, %ID_LISTBOX, Rec
Loop
```

**sIFN**

**sIFN (*rsField as String, [rlSetNumber Long]*) String**

Returns the value for the field passed in ColumnName from the current row for the set number passed in SetNumber.  Due to the frequent usage of this routine it has a very short non-descript name.  If an invalid ColumnName or SetNumber is passed then will exit process.

**Code Example**:

```
Local Rec          as String
Local i            as Long

' Process records
Do While slGetRow
        ' When a row is returned, you use the sIF, sIFN, sIFX, sIFNX functions.
        Rec = sIFN("RowID) & " - " & sIFN("Manuf) & " - " & sIFN("Price)
        Listbox Add hDlg, %ID_LISTBOX, Rec
Loop
```

**sIFX**

**sIFX *(rlColumnNumber Long, [rsModChars String, rlSetNumber Long]) String***

Returns the value for the field passed in ColumnNumber from the current row for the set number passed in SetNumber. Due to the frequent usage of this routine it has a very short non-descript name.

***ModChars:***
- Em = Return errors. This will override the global return errors flag.
- m is the optional message display modifier and can be:
  - 0 = No message is displayed. This is the default.
  - 1 = Display a warning message with OK button. Error is returned when OK pressed.
  - 2 = Display a question message with OK and Cancel buttons. If they press OK, error is returned. If they press Cancel, will exit process. If an error occurs then the return value will be an empty string.
- e = Do not return errors, display message and exit process. This will override the global return errors flag.
- N = Return NULL fields as $NUL. CAUTION: You can not distinguish between a true NULL field and one that contains a single $NUL char.
- D = Decrypt.
- U = Uncompress.
- t = If field is DateTime then do not return time.
- z = If field is DateTime then do not return time if zero.
- d = If field is DateTime then do not return date.

***Note:***
If both ModChars - D and U - are used then it will first decrypt and then uncompress.

***CAUTION:***
Requesting to uncompress a field that is not compressed will cause unpredicable results!

**sIFNX**

**sIFNX** *(rsField as String, [rsModChars String, rlSetNumber Long]) String*

Returns the value for the field passed in ColumnName from the current row for the set number passed in SetNumber. Due to the frequent usage of this routine it has a very short non-descript name.

*ModChars:*
- E = Return errors. If an error occurs then the return value will
- e = Do not return errors even if global ReturnAllErrors flag is on.be an empty string.
- e = Do not return errors even if global ReturnAllErrors flag is on.
- N = Return NULL fields as $NUL. CAUTION: You can not distinguish between a true NULL field and one that contains a single $NUL char.
- D = Decrypt.
- U = Uncompress.
- t = If field is DateTime then do not return time.
- z = If field is DateTime then do not return time if zero.
- d = If field is DateTime then do not return date.

*CAUTION:*
Requesting to uncompress a field that is not compressed will cause unpredicable results!

## slGetChangeCount

## slGetChangeCount *([rsModChars String]) Long*

Returns the number of rows that were changed.  If no ModChar T then returns the number of rows that were changed or inserted or deleted by the most recently completed SQL statement.  Changes caused by triggers are not counted.  If ModChar T then returns the number of row changes since open. Changes caused by triggers are included.


**ModChars**:
- **Em** = Return errors. This will override the global return errors flag.
    m is the optional message display modifier and can be:
        **0** = No message is displayed.  This is the default.
        **1** = Display a warning message with OK button.  Error is returned when OK pressed.
        **2** = Display a question message with OK and Cancel buttons. If they press OK, error is returned.  If they press Cancel, will exit process.
- **e** = Do not return errors, display message and exit process. This will override the global return errors flag.
- **T** = Total changed since open.

Returns zero or the change count if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return -1.

## slGetColumnCount

## slGetColumnCount *([rlSetNumber Long]) Long*

Returns the number of columns in set passed in SetNumber. Will return zero if an error occurs and the global return errors flag is on.

**Code Example**:

```
Local NumFlds      as String
Local Rec          as String
Local i            as Long

'get the number of fields (or columns)
NumFlds = slGetColumnCount

' Process records
Do While slGetRow
        ' When a row is returned, you use the slF, slFN, slFX, slFNX functions.
        Rec = ""
        For i = 1 To NumFlds
                Rec = Rec & slF(i) & " - "
        Next
        Listbox Add hDlg, %ID_LISTBOX, Rec
Loop
```

# slGetColumnName

## slGetColumnName *([rlColumn Long, rlSetNumber Long]) String*

Returns either the name of a column or all column names contained in the set passed in SetNumber.  Will return all column names if the passed column number is omitted or zero. If all column names are returned then it is a delimited text string which is $NUL separated.If ColumnNumber or SetNumber is invalid then will return an empty string.  You may call slGetError or slGetErrorNumber to determine the reason.


**Code Example:**

```
'This can be used to build a header string for Listview or grid
'Notice: If you want to retrieve a whole table then the better way to do this is to
use slGetTableColumnNames.

Local i as Long

For i = 1 to slGetColumnCount
        LISTVIEW INSERT COLUMN hwnd, cID, i, slGetColumnName(i), 70, 0
Next
```

## slGetColumnNumber

## slGetColumnNumber *(rsColumnName String, [rlSetNumber Long]) Long*

Returns the number of the column for the name that is passed in ColumnName contained in the set passed in SetNumber. Will return zero if an error occurs and the global return errors flag is on.

### Code Example:

'This retrieves the column number of a field that has the name 'Field10'.
a& = **slGetColumnNumber** ("Field10")

## slGetError

## slGetError *() String*

Returns the error number and description of the last error that occurred. Each command will initialize/set this value so you must call slGetError before doing another command. If error number is a minus value then it is from SQLitening, if positive value then it is from SQLite3. The returning format is: number = description

**Code Example**:

Local lErr as String

slOpen "sample.db3", "E0"

lErr = **slGetError**
If Val(lErr) Then
        MsgBox lErr, 0, "Database Error"
End If

## slGetErrorNumber

## slGetErrorNumber *() Long*

Returns the error number of the last error that occurred. Each command will initialize/set this value so you must call slGetErrorNumber before doing another command. If error number is a minus value then it is from SQLitening, if positive value then it is from SQLite3.

**Code Example**:

Local lErNumr as Long

slOpen "sample.db3", "E0"

lErrNum = **slGetErrorNumber**
If lErrNum <> 0 then
    MsgBox "Error Number is: " & Str$(lErrNum), 0, "Error"
End If

**slGetFile**


## slGetFile *(rsFileName String, wsFileData String, [rsModChars String]) Long*

FileName is the flat file (standard or Ini) to process. FileName may also contain an optional password. This password is separated from the file name by the $BS character. If it's an Ini file then append a $VT followed by the optional section name and entity name, $BS delimited. If section name is empty then will return all section names, $NUL delimited. If entity name is empty then will return all entity names, $NUL delimited. If entity name is not empty then will return the value for that entity.

### Examples:
1. filename [$BS password] -- will get standard file data
2. filename [$BS password] $VT -- will get all Ini section names
3. filename [$BS password] $VT sectionname -- will get all Ini entity names
4. filename [$BS password] $VT sectionname $BS entityname -- will get value for Ini entity

### ModChars:
- Em = Return errors. This will override the global return errors flag.
- m is the optional message display modifier and can be:
  - 0 = No message is displayed. This is the default.
  - 1 = Display a warning message with OK button. Error is returned when OK pressed.
  - 2 = Display a question message with OK and Cancel buttons. If they press OK, error is returned. If they press Cancel, will exit process.
- e = Do not return errors, display message and exit process. This will override the global return errors flag.
- S = Open file with Shared lock, default is both Read and Write locks. This only applies to standard files, not Ini.

Returns zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

### --- Local Mode ---
If the FileName is not fully pathed then it is assumed to be relative to the current folder which is the same folder, unless changed by chdir, that the first .Exe was run from. If fully pathed then no assumptions are made. Since this running on a local computer SQLitening allows the files to be located anywhere on the local hard drives or local network drives.

### Examples:
1. If your .Exe started in C:\Apps\MyApp then:
2. If FileName is X\Y\Able.Sld then will assume file is in C:\Apps\MyApp\X\Y.

3. If FileName is ..\Y\Able.Sld then will assume file is in C:\Apps\Y.
4. If FileName is C:\Able.Sld then no assumption.

**--- Remote Mode ---**

The FileName is assumed to be relative to the folder which the service is running from. Since this is running on a remote server SQLitening can not allow the user to access files anyplace on the server. Access is denied to any FileName that has a colon, a double dot, or begins with a backslash. This will insure that the file is in same folder as the service or below it. Access is always denied to SQLiteningServer.Cfg. This is for security reasons since that is where passwords are stored. Also for security reasons, all files must pre-exist.

*Examples:*

If your service is running from C:\SQLitening then:
1. If FileName is Data\Able.Sld then will assume file is in C:\SQLitening\Data.
2. If FileName is ..\Y\Able.Sld then will get error -8 Access Denied.

If FileName is C:\Able.Sld then will get error -8 Access Denied.

## slGetHandle

## slGetHandle *([rsModChars String, rlSetNumber Long]) Dword*

Returns the requested handle.  ModChars will determine which handle is returned.  These handles can be used to call SQLite directly but only in local mode.  Raises error %SQLitening_InvalidStringOrRequest if in remote mode

**ModChars:**
- D = Return the open database handle. This is the default.
- S = Return the open set handle for set number passed in SetNumber.

Returns zero if an error occurs and the global return errors flag is on.

**slGetInsertID**

## slGetInsertID *() Quad*

Returns the rowid key of the most recent insert into the database. Returns zero if an error occurs and the global return errors flag is on.

**slGetRow**


**slGetRow *([rlSetNumber Long, rsModChars String]) Long***

Gets the next selected row of fields for set passed in SetNumber. Returns %True if there is a next row available. Returns %False if no row is available (end of set) and will close the set. If an error occurs then, depending on ModChars and the global return errors flag,
will either display error and exit or will return %False.

*ModChars:*
- Em = Return errors. This will override the global return errors flag.
- m is the optional message display modifier and can be:
    - 0 = No message is displayed. This is the default.
    - 1 = Display a warning message with OK button. Error is returned when OK pressed.
    - 2 = Display a question message with OK and Cancel buttons. If they press OK, error is returned. If they press
      Cancel, will exit process.
- e = Do not return errors, display message and exit process. This will override the global return errors flag.
- C = Close the set even if there is another row. This is useful when all you want to do is test if a set has any rows.


**Code Example**:

```
Local Rec            as String

' Process records
Do While slGetRow
    ' When a row is returned, you use the slF, slFN, slFX, slFNX functions.
    Rec = slFN("RowID) & " - " & slFN("Manuf) & " - " & slFN("Price)
    Listbox Add hDlg, %ID_LISTBOX, Rec
Loop
```

## slGetStatus (rlRequest Long) String

Returns the requested status which is normally a $NUL delimited list. If an error occurs then will return an empty string. You may call slGetError or slGetErrorNumber to determine the reason.

The following requests are currently valid:

**1 = Return all named locks.** Will always return an empty string if running in local mode. Each named lock will be delimited by vertical tabs ($VT). Lock data will be delimited by backspaces ($BS) as follows:
>      1 = Tcp file number that owns the lock
>      2 = Lock value
>      3 = Status (0 = Locked, <>0 = Waiting)
>      4 = Time lock set (Milliseconds after midnight)
>      5 = User
>      6 = Computer
>      7 = IPAddress
>      8 = Connect Date-Time

**2 = Return all connections.** Will always return an empty string if running in local mode. Each connection will be delimited by vertical tabs ($VT). Connection data will be delimited by backspaces ($BS) as follows:
>      1 = Tcp file number
>      2 = User
>      3 = Computer
>      4 = IPAddress
>      5 = Connect Date-Time

**3 = Return SQLitening flag settings.** Will return a comma delimited list of one or zero for each flag as follows:
>      1 = One if AreRunningRemote is on.
>      2 = One if AreConnected is on.
>      3 = One if ReturnAllErrors is on.
>      4 = One if DisplayErrorMessage is on.
>      5 = One if DisplayErrorMessageQuestion is on.
6 = One if DoNotRetryIfBusy is on.

**slGetTableColumnNames**

**slGetTableColumnNames** *(rsTableName String) String*

Returns a list of column names contained in passed TableName. The names are returned as a delimited text string which is $NUL separated. If the table name is invalid then will return an empty string. No error will occur.


**Code Example:**

```
'This can be used to build a header string for Listview or grid
'Notice: For retrieving the column names of a SELECT use slGetColumnName
'Instead of ParseCount slGetColumnCountcould also be used.

Local i as Long
Local cNames as String

cNames = slGetTableColumnNames("Parts")

For i = 1 to ParseCount(cNames, Chr$(0)
      LISTVIEW INSERT COLUMN hDlg, cID, i, Parse$(cNames, Chr$(0), i),
70, 0
Next
```

**slGetTableNames**

**slGetTableNames ([rsDataBase String]) String**

Returns a list of table names contained in passed DataBase. The names are returned as a delimited text string which is $NUL separated.  If DataBase is omitted or empty then will default to Main. The DataBase parm must be passed to get the table names of attached databases.  If an error occurs then will return an empty string.  You may call slGetError or slGetErrorNumber to determine the reason.  Returns an empty string if an error occurs and the global return errors flag is on.


**Code Example:**

```
Local tblNames      as String
Local tblNum        as Long
Local i             as Long

tblNames = slGetTableNames("sample.db3")

'Get number of tabel
tblNum = ParseCount(tblNames, Chr$(0))

For i = 1 to tblNum
      Listbox Add hDlg, %ID_LISTBOX, Parse$(tblNames, Chr$(0), i)
Next
```

## slGetUnusedSetNumber

## slGetUnusedSetNumber *() Long*

Will return the lowest set number currently not being used. No error can occur.

## sIIsColumnNameValid

## sIIsColumnNumberValid *(rlColumnNumber Long, [rlSetNumber Long]) Long*

Returns %True if ColumnNumber is valid. Returns %False if it is not valid. If an error occurs then, depending on the global return errors flag, will either display error and exit or will return %False.

## sIIsColumnNumberValid

## sIIsColumnNumberValid *(rlColumnNumber Long, [rlSetNumber Long]) Long*

Returns %True if ColumnNumber is valid. Returns %False if it is not valid. If an error occurs then, depending on the global return errors flag, will either display error and exit or will return %False.

## slIsFieldNull

## slIsFieldNull *(rlColumnNumber Long, [rlSetNumber Long]) Long*

Returns %True if the type of field for the column passed in ColumnNumber is Null. Returns %False if it is not Null. If an error occurs then, depending on the global return errors flag, will either display the error and exit or will return %False.

**slIsOpen**

**slIsOpen () Long**

Returns %TRUE if there is a database open.

## sIIsSetNumberValid

## sIIsSetNumberValid *([rlSetNumber Long]) Long*

Returns %True if SetNumber is valid or %False if SetNumber is invalid (not in array or closed). No error can occur.

**slOpen**

## slOpen *([rsFileName String, rsModChars String]) Long*

FileName is the SQLite file to open as a database. There are two special file names:
1. :memory: = opens an empty database in memory
2. empty or omitted = opens an empty temporary database on disk

FileName may also contain an optional password. This password is separated from the file name by the $BS character.

### ModChars:
- C = Create if not there. Ignored if ReadOnly.
- Em = Return errors. This will override the global return errors flag.
- m is the optional message display modifier and can be:
  - 0 = No message is displayed. This is the default.
  - 1 = Display a warning message with OK button. Error is returned when OK pressed.
  - 2 = Display a question message with OK and Cancel buttons. If they press OK, error is returned. If they press Cancel, will exit process.
- e = Do not return errors, display message and exit process. This will override the global return errors flag.
- R = ReadOnly.
- Tn = Where n is milliseconds to wait for a database lock. Defalut is 10000 milliseconds (10 seconds). If running in remote mode then the TCP SEND/RECV wait as set in slConnect should be at least 3 times as long as this wait. Returns zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

### --- Local Mode ---
If the FileName is not fully pathed then it is assumed to be relative to the current folder which is the same folder, unless changed by chdir, that the first .Exe was run from. If fully pathed then no assumptions are made. Since this running on a local computer SQLitening allows the files to be located anywhere on the local hard drives or local network drives.

### Examples:
If your .Exe started in C:\Apps\MyApp then:
1. If FileName is X\Y\Able.Sld then will assume file is in C:\Apps\MyApp\X\Y.
2. If FileName is ..\Y\Able.Sld then will assume file is in C:\Apps\Y.
3. If FileName is C:\Able.Sld then no assumption.

### --- Remote Mode ---

The FileName is assumed to be relative to the folder which the service is running from.  Since this is running on a remote server SQLitening can not allow the user to access files anyplace on the server.  Access is denied to any FileName that has a colon, a double dot, or begins with a backslash.  This will insure that the file is in same folder as the service or below it.

**Examples:**
If your service is running from C:\SQLitening then:
1. If FileName is Data\Able.Sld then will assume file is in C:\SQLitening\Data.
2. If FileName is ..\Y\Able.Sld then will get error -8 Access Denied.
3. If FileName is C:\Able.Sld then will get error -8 Access Denied.
4. If FileName is X\Y\Able.Exe and command was GetFile then will get error -8 (Access Denied).


**Code Examples**:

```
'Open an existing database
Local lErr as String

slOpen "sample.db3", "E0"

lErr = slGetError
If Val(lErr) Then
      MsgBox lErr, 0, "Database Error"
End If


'This creates a new empty database or open it in case it does exist already.
slOpen "MyDatabase.db3", "C"
```

How to create a table see example in slExe.

## slPopDatabase

## slPopDatabase *[rsSave String]*

Will pop (restore) the database handle and database flags from either the internal stack or from passed Save. If Save is omitted then data will be obtained from the internal stack. If Save is passed then if must be formated same as in slPushDatabase. No error can occur.

**slPopSet**

**slPopSet [rlSetNumber Long, rsSave String]**

Will pop(restore) the set data from either the internal stack or from the passed Save.  If Save is omitted or zero then the data will be obtained from the internal stack.  If an invalid SetNumber is passed then will exit process.

## slPushDatabase

## slPushDatabase *[wsSave String]*

Will push(save) the database handle and database flags to either the internal stack or return it in passed Save. If Save is omitted then the handle will be placed on the internal stack. The current database handle is forgotten but not closed. A slPopDatabase is required before reusing this database. No error can occur.

## slPushSet

## slPushSet [rlSetNumber Long, wsSave String]

Will push (save) the data about an open set to either the internal stack or return it in the passed Save. If Save is omitted then the data will be placed on the internal stack. A slPopSet is required before reusing this set. If an invalid SetNumber is passed then will exit process.

## slPutFile

## slPutFile *(rsFileName String, rsFileData String, [rsModChars String]) Long*

FileName is the flat file (standard or Ini) to process. FileName may also contain an optional password. This password is separated from the file name by the $BS character. If it's an Ini file then append a $VT followed by the section name and entity name, $BS delimited. The section name is required. If entity name is empty then the section wil be deleted. FileData will contain either the standard file data or an Ini entity value. If the Ini entity value in FileData is ommitted then the entity entry, name and value, will be deleted.

### *Examples*
1. filename [$BS password] -- will put standard file data
2. filename [$BS password] $VT sectionname -- sectionname will be deleted
3. filename [$BS password] $VT sectionname $BS entityname -- if FileData if empty then will delete the entityname entry else will add or update the entityname entry.

### *ModChars:*
- C = Create if not there. Ignored if running remote.
- Em = Return errors. This will override the global return errors flag.
- m is the optional message display modifier and can be:
    - 0 = No message is displayed. This is the default.
    - 1 = Display a warning message with OK button. Error is returned when OK pressed.
    - 2 = Display a question message with OK and Cancel buttons. If they press OK, error is returned. If they press Cancel, will exit process.
- e = Do not return errors, display message and exit process. This will override the global return errors flag.
- S = Open file with Shared lock, default is both Read and Write locks. This only applies to standard files, not Ini.

Returns zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

### *--- Local Mode ---*
If the FileName is not fully pathed then it is assumed to be relative to the current folder which is the same folder, unless changed by chdir, that the first .Exe was run from. If fully pathed then no assumptions are made. Since this running on a local computer SQLitening allows the files to be located anywhere on the local hard drives or local network drives.

***Examples:***

If your .Exe started in C:\Apps\MyApp then:

1. If FileName is X\Y\Able.Sld then will assume file is in C:\Apps\MyApp\X\Y.
2. If FileName is ..\Y\Able.Sld then will assume file is in C:\Apps\Y.
3. If FileName is C:\Able.Sld then no assumption.

***--- Remote Mode ---***

The FileName is assumed to be relative to the folder which the service is running from.  Since this is running on a remote server SQLitening can not allow the user to access files anyplace on the server.  Access is denied to any FileName that has a colon, a double dot, or begins with a backslash. This will insure that the file is in same folder as the service or below it. Access is denied to files that are in the Temp folder (contains the Row Data Chunk overflows), any file whose extension is Exe, .Dll, .Log or .Cfg that reside in same folder as the service. Also for security reasons, all files must pre-exist and be included in the FACT of the server config file. This is because of the damage that can be done due to a coding error with PutFile.

***Examples:***

If your service is running from C:\SQLitening then:

1. If FileName is Data\Able.Sld then will assume file is in C:\SQLitening\Data.
2. If FileName is ..\Y\Able.Sld then will get error -8 Access Denied.
3. If FileName is C:\Able.Sld then will get error -8 Access Denied.
4. If FileName is Able.Exe then will get error -8 Access Denied.

If FileName is Temp\Goffy.Sld then will get error -8 Access Denied.

## slRunProc

## slRunProc *(rsProcName String, blParm1 Long, blParm2 Long, bsParm3 String, bsParm4 String, [rsModChars String]) Long*

ProcName is the one character library suffix followed by the name of the entry to be called within the library. ProcName may also contain an optional password. This password is separated from the proc name by the $BS character. The full library name is SQLiteningProcs + the one character suffix. A ProcName of AStoreOrder^TallWalk (^ is the $BS) would have the following three parts:

1. A is the library name suffix so SQLiteningProcsA.Dll would be loaded.
2. StoreOrder is the entry name.
3. TallWalk is the optional password.

Normally when you RunProc the library is loaded, the proc is run, and then the library is unloaded. This is the default, you may override that with the L, U, and u ModChars. Note that a library that creates SQLite custom functions must remain loaded. If you are only loading or unloading a library (L or U ModChar passed) then the ProcName is only the one character library suffix. Note also that if you fail to unload a library and you are running remotely, it will remain loaded in memory until the service is stopped. SQLiteningProcsA.Bas is included as a sample proc library. There are several    sample entries coded there including how to create SQLite cusotm functions.   Some examples are called by this slRunProc while one example is invoked by    using the SQLite load_extension function.

### *ModChars:*
- **Em** = Return errors. This will override the global return errors flag.
       m is the optional message display modifier and can be:
             **0** = No message is displayed.  This is the default.
             **1** = Display a warning message with OK button.  Error is returned when OK pressed.
             **2** = Display a question message with OK and Cancel buttons.
         If they press OK, error is returned.  If they press Cancel, will exit process.
- **e**  = Do not return errors, display message and exit process. This will override the global return errors flag.
- **L**  = Load the library.  Pass only the one character library prefix in ProcName. Useful if you know you will be running procs many times from this library. You must later unload the library.
- **U**  = Unload the library.  Pass only the one character library prefix in ProcName.  Required if your previously loaded the library.
- **u**  = Do not unload the library after running the proc. Pass the one character library prefix and entry name in ProcName. You must later unload the library

using the U ModChar.

## slSel

## slSel *(rsStatement String, [rlSetNumber Long, rsModChars String]) Long*

Will prep the passed Statement and if no errors will activate the passed SetNumber (must have been closed/inactive). This SetNumber can then be used in slGetRow and FieldGet (slF, slFN, slFX, slFNX) commands to get rows and data. The Statement should be Select or Pragma or any other that returns rows (Insert, Update, and Delete do not return rows). SetNumber can be omitted or be any value from 0 to 32767. If omitted then will use zero. Since SetNumber is used as an array index, no gaps is best, which will result in a smaller array. You can have as many unique sets open/active at same time as your memory will allow.

### *ModChars:*

- **D** = Allow duplicate column names. Not recommended if using slFN or slFNX. because you will always get the first value returned. SQLite does not normally return qualified column names. SQLite will return C1 twice if you Select T1.C1, T2.C1. So the solution is to alias one of them with the As clause as follows Select T1.C1, T2.C1 as C1Again. There is a Pragma called "full_column_names" which forces SQLite to return qualified names, but does not seem to work if you Select *. Read up on it and use if you like. I like using an alias because it is less code and more clear.
- **Em** = Return errors. This will override the global return errors flag.
    m is the optional message display modifier and can be:
    - **0** = No message is displayed. This is the default.
    - **1** = Display a warning message with OK button. Error is returned when OK pressed.
    - **2** = Display a question message with OK and Cancel buttons. If they press OK, error is returned. If they press Cancel, will exit process.
- **e** = Do not return errors, display message and exit process. This will override the global return errors flag.
- **Bn** = Do a Begin Transaction before doing the Sel command. The type of Begin is controlled by the value of n as follows:
    - **0** = Deferred. This is the default if n is omitted.
    - **1** = Immediate.
    - **2** = Exclusive.
    This allows for database locking and selecting in one trip to the server. CAUTION: If the Begin or the Select returns Busy then will restart with the Begin. Use Begin Immediate to prevent this or set ProcessMods to %gbfDoNotRetryIfBusy.

- **R** = Release all named locks owned by this connection after doing the Sel.

Returns zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

**Code Example**:

Local SqlExpr as String

'use any like these
SqlExpr = "SELECT * FROM parts"
SqlExpr = "SELECT RowID, * FROM parts WHERE MANUF = '3COM' ORDER BY REDREF"
SqlExpr = "SELECT RowID, Manuf, Procduct, Pgroup, Price ORDER BY Pgroup, Manuf"

**slSel** SqlExpr

## slSelAry

### slSelAry *(rsStatement String, wsaColsAndRows() String, [rsModChars String]) Long*

Loads the passed Data array with all the column data from each row which is returned by the select statement passed in Statement.  The returned array will be one or two dimensions.If the 'Q' ModChar is not passed then the array will be two decisions where the first dimension is equal to the number of columns while the second is equal to the number of rows (plus one if column names are returned).

If the 'Q' ModChar is passed then the array is one dimension with an entry for each row (plus one if column names are returned) and the columns will    be delimited by the ascii value following the 'Q' ModChar. If a 'c' ModChar is not passed then the index of the first row is zero and will contain the column names (lbound = 0).  If a 'c' ModChar is passed then the index of the first row is one and no column names are passed (lbound = 1). Regardless of the 'c' ModChar, the first row data is always index one.

Beware that very large record sets will consume lots of memory.

### *ModChars:*

- **Em** = Return errors. This will override the global return errors flag.
    **m** is the optional message display modifier and can be:
        **0** = No message is displayed.  This is the default.
        **1** = Display a warning message with OK button.  Error is returned when OK pressed.
        **2** = Display a question message with OK and Cancel buttons. If they press OK, error is returned.  If they press Cancel, will exit process.
- **e**  = Do not return errors, display message and exit process. This will override the global return errors flag.
- **Bn** = Do a Begin Transaction before doing the Sel command. The type of Begin is controlled by the value of n as follows:
        **0** = Deferred. This is the default if n is omitted.
        **1** = Immediate.
        **2** = Exclusive.
        This allows for database locking and selecting in one trip to the server.
        **CAUTION**: If the Begin or the Select returns Busy then will restart with the Begin.  Use Begin Immediate to prevent this or set ProcessMods to %gbfDoNotRetryIfBusy.
- **R** =  Release all named locks owned by this connection after doing the SelAry.
- **Q#** = Return a one dimension array where each column is delimited by a

single character.  That character is determined by the ascii value at #. The default is to return a two dimension array.

- **c** = Do not return the column names as a row of data at index zero.  The default is to return the column names as row zero.
- **D$** = Decrypt.
- **U$** = Uncompress.
- **N$** = Return NULL fields as $NUL.  CAUTION: You can not distinguish between a true NULL field and one that contains a single $NUL char.
- **t$** = If field is DateTime then do not return time.
- **z$** = If field is DateTime then do not return time if zero.
- **d$** = If field is DateTime then do not return date.

**Note**: The above $ is a comma delimited list of column numbers. Beware that this columns apply to ALL rows.  So if, for example, sometimes a column is compressed and sometimes not then do not use slSelAry, rather build your own array using slSel.

**Example**:
slSelAry "Select F1, F2 from T1", A(), "D1,2N2" would decrypt columns 1 and 2 and also return $NUL if column 2 is Null. If table T1 had 3 rows then the array dimensions would  be (1 to 2, 0 to 3).  A(1,0) would contain "F1". A(2,0) would contain "F2".  A(1,1) would contain the value of F1 from row 1. A(2,1) would contain the value of F2 from row 1. Etc....

**Example**:
slSelAry "Select F1, F2 from T1", A(), "Q9c" would return a one dimension array with the columns tab delimited and no column names in index zero.

Returns zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

**Code Example**:

```
#Dim All
#Compile Exe
#Include "SQLitening.Inc"

Function PbMain()
        Local st    As String
        Dim a(10) as String
        Local i as Long

        slOpen "sample.db3"
        slSelAry "SELECT * FROM Parts LIMIT 10", a(), "Q9"
```

```
ST = a(0) & $crlf & $crlf
FOR i = 1 to 10
        ST = ST & a(i) & $crlf
NEXT

? st, 0, "slSelAry Test"
End Function
```

## slSetProcessMods

## slSetProcessMods *(rsModChars String)*

Sets the process mod flags, modes, and/or values as controlled by the passed ModChars. Some of the mods are global while some are for the open database. No error can occur.

*ModChars:*
- R = Retry if busy. This is the default. If database is locked then message will display asking if want to retry. A database must be open. This setting is pushed/poped along with then the database handle.
- r = Do not retry if busy. If database is locked no message will display and either an error will be returned or raised. A database must be open. This setting is pushed/poped along with the database handle.
- Em = Return errors. This sets the global return errors flag on. If compiled with the %ReturnAllErrors conditional compile set to %True then this is the default upon start up.
  m is the optional message display modifier and can be:
    - 0 = No message is displayed. This is the default.
    - 1 = Display a warning message with OK button. Error is returned when OK pressed.
    - 2 = Display a question message with OK and Cancel buttons. If they press OK, error is returned. If they press Cancel, will exit process.
- e = Do not return errors, display message and exit process. This sets the global return errors flag off. If compiled with the %ReturnAllErrors conditional compile set to %False then this is the default upon start up.
- Tn = Set SQLite Busy Timeout. n = Number of milliseconds. Default is 10000 or 10 seconds to wait for a database lock. This is originally set in slOpen. If running in remote mode then the TCP SEND/RECV wait as set in slConnect should be at least 3 times as long as this wait. This setting is global.
- Ln = Load either the local or remote processing lib and ruts. This along with push/pop database allows you to access both a local and remote database in the same running program. WARNING: The misuse of this may cause a GPF.
    - n = 0 will load SQLite3.DLL for Local Access Mode
    - n = 1 will load SQLiteningCliend.DLL. for Remote Access Mode

Kx = Set the crypt key for subsequent encryption/decryption. x is the key string, it must immediately follow the K. If there is no value following the K then any existing crypt key will be cleared, this is for security reasons. This ModChar MUST be used alone since the letters in the key may be equal to other ModChars. If you have not written your own custom encryption then the passed key must be either 16, 24, or 32 bytes long.

slSetRelNamedLocks

slSetRelNamedLocks (rsLockNames String, [rsModChars String, rsSelStatement String, rlSelSetNumber as Long, rsSelModChars String) Long

Will set or release named lock(s).  This can be used to implement application locks or to simulate record locks.  Names are locked on a server-wide basis.  If a name has been locked by one client, the server blocks any request by another client for a lock with the same name.  This allows clients that agree on a given lock name to use the name to perform cooperative named locking.  But be aware that it also allows a client that is not among the set of cooperating clients to lock a name and thus prevent any of the cooperating clients from locking that name. One way to reduce the likelihood of this is to use lock names that are database-specific or application-specific.  Named locks are only used in remote mode, ignored when running in local mode.Will also optionally do a Sel command but only if the lock request was successful.

**LockNames**:
This is a vertical tab ($VT) delimited list of names to set or release.Be sure these names uniquely identify the lock you want to set.  If, for example, you know the RowID of a record you want to lock you would need to include some type of table ID within the name. The first character of each name must be one of the following:

> **+** (plus)  = Set lock
> **-** (minus) = Release lock.

All named locks owned by a connection are automatically released when the connection is no longer active.

**ModChars**:
> **Tn** = Where n is the number of milliseconds to wait for a named lock to be set before returning as unsuccessful.  If omitted or is zero then will return immediately if a named lock can not be set because it is owned by another connection.
> **R** = Release all named locks owned by this connection before setting or releasing any named locks in LockNames.

**SelStatement**:
If the SelStatement is not empty and locking was successful then will do the Sel command.  The Statement should be Select or Pragma or any other that returns rows (Insert, Update, and Delete do not return rows).

**SelSetNumber**:
SelSetNumber can be omitted or be any value from 0 to 32767. If omitted then

will use zero.  If SelStatement is empty then this is ignored.SelSetNumber is used as an array index, no gaps is best, which will result in a smaller array.  You can have as many unique sets open/active at same time as your memory will allow.

**SelModChars**:
This value is ignored it SelStatement is empty.

> **D  = Allow duplicate column names.** Not recommended if using slFN or slFNX. because you will always get the first value returned.  SQLite does not normally return qualified column names.  SQLite will return C1 twice if you Select T1.C1, T2.C1. So the solution is to alias one of them with the As clause as follows Select T1.C1, T2.C1 as C1Again. There is a Pragma called "full_column_names" which forces SQLite to return qualified names, but does not seem to work if you Select *. Read up on it and use if you like.  I like using an alias because it  is less code and more clear.

> **Em = Return errors**. This will override the global return errors flag. m is the optional message display modifier and can be:
> > **0** = No message is displayed.  This is the default.
> > **1** = Display a warning message with OK button.  Error is returned when OK pressed.
> > **2** = Display a question message with OK and Cancel buttons.If they press OK, error is returned.  If they press Cancel, will exit process.

> **e  = Do not return errors**, display message and exit process. This will override the global return errors flag.

> **Bn = Do a Begin** Transaction before doing the Sel command. The type of Begin is controlled by the value of n as follows:
> > **0** = Deferred. This is the default if n is omitted.
> > **1** = Immediate.
> > **2** = Exclusive.
> This allows for database locking and selecting in one trip to the server.
> **CAUTION**: If the Begin or the Select returns Busy then will restart with the Begin.  Use Begin Immediate to prevent this or set ProcessMods to %gbfDoNotRetryIfBusy.

> **R =  Release all named locks** owned by this connection after doing the Sel.

**ReturnCode**:
Zero if locking was successful. If locking failed then depending on ModChars and the global return errors flag, will either display error and exit or will return %SQLitening_LockTimeout.  If locking failed then no locks are set nor are any released and no Sel is run. If locking was successful and there is an optional

SelStatement then will return zero if processed OK. Else, depending on ModChars and the global return errors flag, will either display error and exit or will return the error number.

**Examples**:  (No error checking shown)
- slSetRelNamedLocks("+Cust1395" & $VT & +"Cust40928")  --  Will set two named locks.
- slSetRelNamedLocks("+Cust1395" & $VT & "-Cust40928")  --  Will set one named locks and release another.
- slSetRelNamedLocks("-Cust1395")  --  Will release one named locks.
- slSetRelNamedLocks("+Cust1395", "T15000R", "Select * from Cust where Num=1395")
- Will first release all locks for this connection then set one named lock, waiting up to 15 seconds. If lock was set OK then will select the customer.
- slSetRelNamedLocks("", "R")   --  Will release all locks for this connection.

slSetRelNamedLocks("")   --  Will do nothing.

# White Papers

## SQLitening White Papers

[SQLitening API's](#)
[SQLitening Server Log Layout](#)
[SQLitening Error Codes](#)
[SQLitening Aux Ruts](#)

## SQLite White Papers

[SQLite Data Types](#)
[SQLite Error Codes](#)
[SQLite Expressions](#)
[SQLite Functions](#)
[SQLite Syntax](#)
[SQLite Pragma](#)
[SQLite Notes](#)
[SQLite Foreign Key](#)
[SQLite Getting Schema Data](#)

## SQLitening API's

SQLitening comes with three different API's as follows:

- **Basic**
  Dll name is SQLitening.Dll
  Include name is SQLitening.Inc
  All routines begin with sl.

- **Special**
  Dll name is SQLiteningS.Dll
  Include name is SQLiteningS.Inc
  All routines begin with sls.

- **Universal**
  Dll name is SQLiteningU.Dll
  Include name is SQLiteningU.Inc
  All routines begin with slu.

All DLL's are written in PowerBASIC and use the SDECL (aka STDCALL) convention for parameter passing. SDECL specifies that the declared procedure uses the "Standard Calling Convention" as defined by Microsoft. When calling an SDECL procedure, parameters are passed on the stack from right to left and the stackl is automatically cleaned before execution returns to the calling code.

PowerBASIC allocates strings using the Win32 OLE string engine. This allows you to pass strings from your program to DLLs, or API calls that support OLE strings.

### Which API or DLL should you use?

**Basic**:: Always try to use the Basic API.  If you language supports passing OLE strings both ByVal and ByRef then you should be able to use the Basic API. OLE strings are allocated using the Win32 OLE string engine.  There are some routines in the Basic API that will probably only work with PowerBASIC.  slSelAry is one that passes an array and therefore may only work in PowerBASIC.  You may need to modify the include file (SQLitening.Inc) to support your language.

**Special**:: If your language supports OLE string passing but only ByRef  (Visual Basic) then the Special API will probable work for you.  The language must also support optional parameter passing.  The slSelAry is not available in this API. You will need to modify the include file (SQLiteningS.Inc) to support your language This DLL is a front-end to the Basic API.

**Universal**:: If your language does not support OLE strings then the Special API will probable work for you.  This API will work for any language that can call a

standard Dll.  The parameter passing is patterned after the Windows API. The slSelAry is not available in this API.  You will need to modify the include file (SQLiteningU.Inc) to support your language This DLL is a front-end to the Basic API.  Documentation about parameter passing is located in the include file.

# SQLitening Server Log Layout

The layout is as follows:

| | | |
|---|---|---|
| 1 - 6 | = | **Date** as YYMMDD |
| 7 - 12 | = | **Time** as HHMMSS |
| 14 - 17 | = | Entry Type as: |

      **Admn** -- followed by a 4 byte message type of Strt, Stop, Flgs, Info, or FACT and then the message.

      **Host** -- followed by the host name, IP address, socket, port, and service name suffix.

      **Conn** -- followed by # and the TCP file number and "SK" and the socket number and user data. User data consists of user name, computes name, and IP address delimited by $BS. Any part of the user data may be omitted by the calling program.

      **Dcon** -- followed by # and the TCP file number and one of the following:
      "Dropped" "Disconnect" "TimeOut" "Killed" "Error"

      **Eror** -- followed by the error message.

      **Exit** -- followed by the custom message from SQLiteningServerExit.Dll

| | | |
|---|---|---|
| 19 - ?? | = | **Data** depending upon Record Type above. |

The log was designed to be as compact as possible for space and speed reasons and also be humanly readable.

The TCP file numbers in Conn, User, and Dcon allow you to match up connects with disconnects and compute elapsed time for a user.

- **1 - 6** = Date as YYMMDD
- **7 - 12** = Time as HHMMSS
- **14 - 17** = Record Type as:Admn -- followed by a 4 byte message type of Strt, Stop, Flgs, or FACT and then the message.
- **Host** -- followed by the host name, IP address, socket, and port.
- **Conn** -- followed by # and the TCP file number and "SK" and the socket number and user data. User data consists of user name, computes name, and IP address delimited by $BS. Any part of the user data may be omitted by the calling program.
- **Dcon** -- followed by # and the TCP file number.

- **Error** -- followed by the error message.

**19 - ??** = Data depending upon Record Type above.

The log was designed to be as compact as possible for space and speed reasons and also be humanly readable.

The TCP file numbers in Conn, User, and Dcon allow you to match up connects with disconnects  and compute elapsed time for a user.

## SQLitening Error Codes

- **%SQLitening_NoOpenDatabase** = -7 - No open database
- **%SQLitening_AccessDenied** = -8 - Access denied
    - Connect was able to do Tcp Open but was unable to "talk" to SQLitening service. Probable caused by some type of firewall setting or an exit has denied connection.
    - Open, Attach, GetFile, or PutFile password check failed or the file is protected.
- **%SQLitening_FileDoesNotExist** = -9- File does not exist
- **%SQLitening_FileOpenGetPutError** = -10 - File error doing open, get, or put
- **%SQLitening_LockTimeout** = -11 - Timeout occurred trying to lock an object or database
- **%SQLitening_NochangedRows** = -12 - No changed rows
- **%SQLitening_InvalidColumnNameNumber** = -13 - Invalid column name or number
- **%SQLitening_InvalidSetNumber** = -14 - Invalid set number
- **%SQLitening_AttachIsInvalid** = -15 - Can not use SQL Attach
- **%SQLitening_CanNotConnect** = -16 - Can not connect
    - Tcp Open failed.

**%SQLitening_InvalidKeyOrNotFound** = -17 - Invalid key or key not found
- Ini GetFile section or entity not found.
- Ini PutFile WritePrivateProfileString winapi failed.
- Encrypt key must be set and must be 16, 24, or 32 bytes.

**%SQLitening_SendOrReceiveError** = -18 - Error sending or receiving message"
- Prior connect failed, server went away, TCP time-out, or message length error

**%SQLitening_InvalidStringOrRequest** = -19 - Invalid string or request
- Uncompressing string which is not compressed.
- String is too big to compress.
- BindDat is invalid.
- RunProc is invalid library or entry.

**%SQLitening_ErrorAtServer** = -20 - An error occured at server -- check server log.

# SQLitening Aux Ruts

Contains auxiliary routines called by SQLitening.Dll. Can also be called from your program. There currently is only one routine available which is AuxRutsA. SQLitening.Dll contains the ruts slSetProcessMods and slConvertDat which allow you to accomplish same think so this only has use in a program that isn't using SQLitening.

## AuxRutsA  (rlAction Long, rsTextIn String, wsTextOut String) Long

Action is:
> 1 = Set crypt key from TextIn. TextOut is not used.  Empty TextIn will remove key, use for security reasons.
> 2 = Encrypt TextIn into TextOut.
> 3 = Decrypt TextIn into TextOut.
> 4 = Compress TextIn into TextOut.
> 5 = Uncompress TextIn into TextOut.
> 6 = Compress and Encrypt TextIn into TextOut.
> 7 = Convert to Hexadecimal from TextIn into TextOut.
> 8 = Decrypt and Uncompress TextIn into TextOut.

Returns zero if OK or an error number -17 or -19 if error occurs.

## SQLite Data Types

One of the great features of SQLite is that you can create columns with no data type, which will result in an affinity of none, by using the following create syntax:

**Create Table T1 (C1, C2, C3)**

There is no advantage in assigning column affinity using the following code:

**Create Table T1 (C1 Integer, C2 Text, C3 Real)**

*The below was taken from http://www.sqlite.org/datatype3.html.*

1. Storage Classes
Each value stored in an SQLite database has one of the following storage classes:

- NULL. The value is a NULL value.
- INTEGER. The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
- REAL. The value is a floating point value, stored as an 8-byte IEEE floating point number.
- TEXT. The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16-LE).
- BLOB. The value is a blob of data, stored exactly as it was input.

Any column except an INTEGER PRIMARY KEY may be used to store any type of value. The exception to this rule is 'Strict Affinity Mode'.

Storage classes are initially assigned as follows:

Values specified as literals as part of SQL statements are assigned storage class TEXT if they are enclosed by single or double quotes, INTEGER if the literal is specified as an unquoted number with no decimal point or exponent, REAL if the literal is an unquoted number with a decimal point or exponent and NULL if the value is a NULL. Literals with storage class BLOB are specified using the X'ABCD' notation.

Values supplied using the sqlite3_bind_* APIs are assigned the storage class that is requested.

2. Column Affinity

The type of a value is associated with the value itself, not with the column or variable in which the value is stored. (This is sometimes called manifest typing.) All other SQL databases engines that we are aware of use the more restrictive

system of static typing where the type is associated with the container, not the value.

SQLite support the concept of "type affinity" on columns. The type affinity of a column is the recommended type for data stored in that column. The key here is that the type is recommended, not required. Any column can still store any type of data, in theory. It is just that some columns, given the choice, will prefer to use one storage class over another. The preferred storage class for a column is called its "affinity".

Each column in SQLite is assigned one of the following type affinities:

- TEXT
- NUMERIC
- INTEGER
- NONE

A column with TEXT affinity stores all data using storage classes NULL, TEXT or BLOB. If numerical data is inserted into a column with TEXT affinity it is converted to text form before being stored.

A column with NUMERIC affinity may contain values using all five storage classes. When text data is inserted into a NUMERIC column, an attempt is made to convert it to an integer or real number before it is stored. If the conversion is successful, then the value is stored using the INTEGER or REAL storage class. If the conversion cannot be performed the value is stored using the TEXT storage class. No attempt is made to convert NULL blob values.

A column that uses INTEGER affinity behaves in the same way as a column NUMERIC affinity, except that if a real value with no floating point (or text value that converts to such) is inserted it is converted to an integer and stored using the INTEGER storage class.

A column with affinity NONE does not prefer one storage class over It makes no attempt to coerce data before it is inserted.

3. Determination Of Column Affinity

The type affinity of a column is determined by the declared type of the column when the table is created, according to the following rules:

If the datatype contains the string "INT" then it is assigned INTEGER affinity.

If the datatype of the column contains any of the strings "CHAR", "CLOB", or "TEXT" then that column has TEXT affinity. Notice that the type VARCHAR contains the string "CHAR" and is thus assigned TEXT affinity.

If the datatype for a column contains the string "BLOB" or if no datatype is specified then the column has affinity NONE.

Otherwise, the affinity is NUMERIC.

## SQLite Error Codes

- 0 = Successful result
- 1 = SQL error or missing database
- 2 = Internal logic error in SQLite
- 3 = Access permission denied
- 4 = Callback routine requested an abort
- 5 = The database file is locked
- 6 = A table in the database is locked
- 7 = A malloc() failed
- 8 = Attempt to write a readonly database
- 9 = Operation terminated by sqlite3_interrupt()
- 10 = Some kind of disk I/O error occurred
- 11 = The database disk image is malformed
- 12 = NOT USED. Table or record not found
- 13 = Insertion failed because database is full
- 14 = Unable to open the database file
- 15 = NOT USED. Database lock protocol error
- 16 = Database is empty
- 17 = The database schema changed
- 18 = String or BLOB exceeds size limit
- 19 = Abort due to constraint violation (Normally caused by trying to add a non unique key to an index during an Insert or Update)
- 20 = Data type mismatch
- 21 = Library used incorrectly
- 22 = Uses OS features not supported on host
- 23 = Authorization denied
- 24 = Auxiliary database format error
- 25 = 2nd parameter to sqlite3_bind out of range
- 26 = File opened that is not a database file
- 100 = sqlite_step() has another row ready

101 = sqlite_step() has finished executing

## SQLite Expressions

expr ::=  expr binary-op expr |
          expr [NOT] like-op expr [ESCAPE expr] |
          unary-op expr |
          ( expr ) |
          column-name |
          table-name . column-name |
          database-name . table-name . column-name |
          literal-value |
          parameter |
          function-name ( expr-list | * ) |
          expr ISNULL |
          expr NOTNULL |
          expr [NOT] BETWEEN expr AND expr |
          expr [NOT] IN ( value-list ) |
          expr [NOT] IN ( select-statement ) |
          expr [NOT] IN [database-name .] table-name |
          [EXISTS] ( select-statement ) |
          CASE [expr] ( WHEN expr THEN expr )+ [ELSE expr] END |
          CAST ( expr AS type )
like-op ::= LIKE | GLOB | REGEXP

This section is different from the others.  Most other sections of this document talks about a particular SQL command.  This section does not talk about a standalone command but about "expressions" which are subcomponents  of most other commands.

SQLite understands the following binary operators, in order from highest to lowest precedence:

```
||
*   /   %
+   -
<<  >>  &   |
<   <=  >   >=
=   ==  !=  <>  IN
AND
OR
```

Supported unary operators are these:

```
-   +   !   ~
```

Note that there are two variations of the equals and not equals operators. Equals can be either = or ==.  The non-equals operator can be either != or  <>.

The || operator is "concatenate" - it joins together the two strings of its operands. The operator % outputs the remainder of its left operand modulo its right operand.

The result of any binary operator is a numeric value, except for the || concatenation operator which gives a string result.

A literal value is an integer number or a floating point number. Scientific notation is supported. The "." character is always used as the decimal point even if the locale setting specifies "," for this role - the use of "," for the decimal point would result in syntactic ambiguity. A string constant is formed by enclosing the string in single quotes ('). A single quote within the string can be encoded by putting two single quotes in a row - as in Pascal. C-style escapes using the backslash character are not supported because they are not standard SQL. BLOB literals are string literals containing hexadecimal data and preceded by a single "x" or "X" character. For example:

    X'53514697465'

A literal value can also be the token "NULL".

A parameter specifies a placeholder in the expression for a literal value that is filled in at runtime using the sqlite3_bind API. Parameters can take several forms:

> **?NNN** - A question mark followed by a number NNN holds a spot for the NNN-th parameter. NNN must be between 1 and 999.
> **?** - A question mark that is not followed by a number holds a spot for the next unused parameter.
> **:AAAA** - A colon followed by an identifier name holds a spot for a named parameter with the name AAAA. Named parameters are also numbered. The number assigned is the next unused number. To avoid confusion, it is best to avoid mixing named and numbered parameters.
> **$AAAA** - A dollar-sign followed by an identifier name also holds a spot for a named parameter with the name AAAA. The identifier name in this case can include one or more occurances of "::" and a suffix enclosed in "(...)" containing any text at all. This syntax is the form of a variable name in the Tcl programming language.

Parameters that are not assigned values using sqlite3_bind are treated as NULL.

The LIKE operator does a pattern matching comparison. The operand to the right contains the pattern, the left hand operand contains the string to   match against the pattern. A percent symbol % in the pattern matches any sequence of zero or more characters in the string. An underscore _ in the pattern matches any single character in the string. Any other character matches itself or it is

lower/upper case equivalent (i.e. case-insensitive matching). (A bug: SQLite only understands upper/lower case for 7-bit Latin characters. Hence the LIKE operator is case sensitive for 8-bit iso8859 characters or UTF-8 characters. For example, the expression 'a' LIKE 'A' is TRUE but 'æ' LIKE 'Æ' is FALSE.).

If the optional ESCAPE clause is present, then the expression following the ESCAPE keyword must evaluate to a string consisting of a single character. This character may be used in the LIKE pattern to include literal percent or underscore characters. The escape character followed by a percent symbol, underscore or itself matches a literal percent symbol, underscore or escape character in the string, respectively. The infix LIKE operator is implemented by calling the user function like(X,Y).

The LIKE operator is not case sensitive and will match upper case characters on one side against lower case characters on the other. (A bug: SQLite only understands upper/lower case for 7-bit Latin characters. Hence the LIKE operator is case sensitive for 8-bit iso8859 characters or UTF-8 characters. For example, the expression 'a' LIKE 'A' is TRUE but 'æ' LIKE 'Æ' is FALSE.).

The infix LIKE operator is implemented by calling the user function like(X,Y). If an ESCAPE clause is present, it adds a third parameter to the function call. If the functionality of LIKE can be overridden by defining an alternative implementation of the like() SQL function.

The GLOB operator is similar to LIKE but uses the Unix file globbing syntax for its wildcards. Also, GLOB is case sensitive, unlike LIKE. Both GLOB and LIKE may be preceded by the NOT keyword to invert the sense of the test. The infix GLOB operator is implemented by calling the user function glob(X,Y) and can be modified by overriding that function.

The REGEXP operator is a special syntax for the regexp() user function. No regexp() user function is defined by default and so use of the REGEXP operator will normally result in an error message. If a user-defined function named "regexp" is defined at run-time, that function will be called in order to implement the REGEXP operator.

A column name can be any of the names defined in the CREATE TABLE statement or one of the following special identifiers: "ROWID", "OID", or "_ROWID_". These special identifiers all describe the unique random integer key (the "row key") associated with every row of every table. The special identifiers only refer to the row key if the CREATE TABLE statement does not define a real column with the same name. Row keys act like read-only columns. A row key can be used anywhere a regular column can be used, except that you cannot change the value of a row key in an UPDATE or INSERT statement. "SELECT * ..." does not return the row key.

SELECT statements can appear in expressions as either the right-hand operand of the IN operator, as a scalar quantity, or as the operand of an EXISTS operator.  As a scalar quantity or the operand of an IN operator, the SELECT should have only a single column in its result.  Compound SELECTs (connected with keywords like UNION or EXCEPT) are allowed.  With the EXISTS operator, the columns in the result set of the SELECT are ignored and the expression returns TRUE if one or more rows exist and FALSE if the result set is empty.  If no terms in the SELECT expression refer to value in the containing query, then the expression is evaluated once prior to any other processing and the result is reused as necessary. If the SELECT expression does contain variables from the outer query, then the SELECT is reevaluated every time it is needed.

When a SELECT is the right operand of the IN operator, the IN operator   returns TRUE if the result of the left operand is any of the values generated by the select.  The IN operator may be preceded by the NOT keyword to invert the sense of the test.

When a SELECT appears within an expression but is not the right operand of an IN operator, then the first row of the result of the SELECT becomes the value used in the expression.  If the SELECT yields more than one result row, all rows after the first are ignored.  If the SELECT yields no rows, then the value of the SELECT is NULL.

A CAST expression changes the datatype of the into the type specified by <type>.  <type> can be any non-empty type name that is valid for the type in a column definition of a CREATE TABLE statement.

Both simple and aggregate functions are supported.  A simple function can be used in any expression.  Simple functions return a result immediately based on their inputs.  Aggregate functions may only be used in a SELECT statement. Aggregate functions compute their result across all rows of the result set.

The functions shown below are available by default.  Additional functions may be written in C and added to the database engine using the sqlite3_create_function() API.  See Functions for details
> abs(X)
> coalesce(X,Y,...)
> glob(X,Y)
> ifnull(X,Y)
> last_insert_rowid()
> length(X)
> like(X,Y [,Z])
> lower(X)
> max(X,Y,...)
> min(X,Y,...)

nullif(X,Y)
quote(X)
random(*)
round(X)
round(X,Y)
soundex(X)
sqlite_version(*)
substr(X,Y,Z)
typeof(X)
upper(X)

The aggregate functions shown below are available by default.  Additional aggregate functions written in C may be added using the sqlite3_create_function() API.  See Functions for details

avg(X)
count(X)
count(*)
max(X)
min(X)
sum(X)

## SQLite Functions

The functions shown below are available by default. Additional functions may be written in C and added to the database engine using the sqlite3_create_function() API.

**abs**(X) Return the absolute value of argument X.

**coalesce**(X,Y,...) Return a copy of the first non-NULL argument. If all arguments are NULL then NULL is returned. There must be at least 2 arguments.

**glob**(X,Y) This function is used to implement the "X GLOB Y" syntax of SQLite. The sqlite3_create_function() interface can be used to override this function and thereby change the operation of the GLOB operator.

**ifnull**(X,Y) Return a copy of the first non-NULL argument. If both arguments are NULL then NULL is returned. This behaves the same as coalesce() above.

**last_insert_rowid**() Return the ROWID of the last row insert from this connection to the database. This is the same value that would be returned from the sqlite_last_insert_rowid() API function.

**length**(X) Return the string length of X in characters. If SQLite is configured to support UTF-8, then the number of UTF-8 characters is returned, not the number of bytes.

**like**(X,Y [,Z]) This function is used to implement the "X LIKE Y [ESCAPE Z]" syntax of SQL. If the optional ESCAPE clause is present, then the user-function is invoked with three arguments. Otherwise, it is invoked with two arguments only. The sqlite_create_function() interface can be used to override this function and thereby change the operation of the LIKE operator. When doing this, it may be important to override both the two and three argument versions of the like() function. Otherwise, different code may be called to implement the LIKE operator depending on whether or not an ESCAPE clause was specified.

**lower**(X) Return a copy of string X will all characters converted to lower case. The C library tolower() routine is used for the conversion, which means that this function might not work correctly on UTF-8 characters.

**max**(X,Y,...) Return the argument with the maximum value. Arguments may be strings in addition to numbers. The maximum value is determined by the usual sort order. Note that max() is a simple function when it has 2 or more arguments but converts to an aggregate function if given only a single argument.

**min**(X,Y,...) Return the argument with the minimum value. Arguments may be strings in addition to numbers. The minimum value is determined by the usual

sort order. Note that min() is a simple function when it has 2 or more arguments but converts to an aggregate function if given only a single argument.

**nullif**(X,Y) Return the first argument if the arguments are different, otherwise return NULL.

**quote**(X) This routine returns a string which is the value of its argument suitable for inclusion into another SQL statement. Strings are surrounded by single-quotes with escapes on interior quotes as needed. BLOBs are encoded as hexadecimal literals. The current implementation of VACUUM uses this function. The function is also useful when writing triggers to implement undo/redo functionality.

**random**(*) Return a random integer between -2147483648 and +2147483647.

**round**(X)

**round**(X,Y) Round off the number X to Y digits to the right of the decimal point. If the Y argument is omitted, 0 is assumed.

**soundex**(X) Compute the soundex encoding of the string X. The string "?000" is returned if the argument is NULL. This function is omitted from SQLite by default. It is only available the -DSQLITE_SOUNDEX=1 compiler option is used when SQLite is built.

**sqlite_version**(*) Return the version string for the SQLite library that is running. Example: "2.8.0"

**substr**(X,Y,Z) Return a substring of input string X that begins with the Y-th character and which is Z characters long. The left-most character of X is number 1. If Y is negative the the first character of the substring is found by counting from the right rather than the left. If SQLite is configured to support UTF-8, then characters indices refer to actual UTF-8 characters, not bytes.

**typeof**(X) Return the type of the expression X. The only return values are "null", "integer", "real", "text", and "blob". SQLite's type handling is explained in Datatypes in SQLite Version 3.

**upper**(X) Return a copy of input string X converted to all upper-case letters. The implementation of this function uses the C library routine toupper() which means it may not work correctly on UTF-8 strings.

In any aggregate function that takes a single argument, that argument can be preceded by the keyword DISTINCT. In such cases, duplicate elements are filtered before being passed into the aggregate function. For example, the function "count(distinct X)" will return the number of distinct values of column X

instead of the total number of non-null values in column X.

**avg**(X) Return the average value of all non-NULL X within a group. Non-numeric values are interpreted as 0.

**count**(X)

**count**(*) The first form return a count of the number of times that X is not NULL in a group. The second form (with no argument) returns the total number of rows in the group.

**max**(X) Return the maximum value of all values in the group. The usual sort order is used to determine the maximum.

**min**(X) Return the minimum non-NULL value of all values in the group. The usual sort order is used to determine the minimum. NULL is only returned if all values in the group are NULL.

**sum**(X) Return the numeric sum of all numeric values in the group. If there are no input rows or all values are NULL, then NULL is returned. NULL is not a helpful result in that case (the correct answer should be zero) but it is what the SQL standard requires and how most other SQL database engines operate so SQLite does it that way in order to be compatible. You will probably want to use "coalesce(sum(X),0)" instead of just "sum(X)" to work around this design problem in the SQL language.

## SQLite Syntex

CREATE [TEMP] TABLE [If Not Exists] [database-name.]table-name (name [type][column-constraint] [,name [type][column-constraint]][,constraint])

    column-constraint ::=
        PRIMARY KEY [sort-order] [ conflict-clause ] [AUTOINCREMENT] |
        UNIQUE
        NOT NULL
        DEFAULT value
        COLLATE collation-name

    constraint ::=
        PRIMARY KEY ( column-list )
        UNIQUE ( column-list )

    collation-name ::=
        BINARY - Compares string data using memcmp(), regardless of
        text encoding. This is the default.
        REVERSE - Collate in the reverse order to BINARY.
        NOCASE - The same as binary, except the 26 upper case
        characters used by the English language are folded to their lower
        case equivalents before the comparison is performed.

DELETE FROM [database-name.]table-name [WHERE expr]

UPDATE [database-name.]table-name SET assignment[, assignment*] [WHERE expr]

INSERT INTO [database-name.]table-name [(column[, column*]] VALUES (value[, value*])

SELECT [DISTINCT] result [FROM [database-name.]table-name] [WHERE expr] [ORDER BY expr] [LIMIT int]

DROP TABLE [If Exists] [database-name.] table-name

BEGIN [ DEFERRED | IMMEDIATE | EXCLUSIVE ] [TRANSACTION]
END [TRANSACTION] or COMMIT [TRANSACTION]
ROLLBACK [TRANSACTION]
Transactions can be deferred, immediate, or exclusive. Deferred means that no locks are acquired on the database until the database is first accessed. Thus with a deferred transaction, the BEGIN statement itself does nothing. Locks are not acquired until the first read or write operation. The first read operation against a database creates a SHARED lock and the first write operation creates a RESERVED lock. Because the acquisition of locks is deferred until they are

needed, it is possible that another thread or process could create a separate transaction and write to the database after the BEGIN on the current thread has executed.If the transaction is immediate, then RESERVED locks are acquired on all databases as soon as the BEGIN command is executed, without waiting for the database to be used. After a BEGIN IMMEDIATE, you are guaranteed that no other thread or process will be able to write to the database or do a BEGIN IMMEDIATE or BEGIN EXCLUSIVE. Other processes can continue to read from the database, however. An exclusive transaction causes EXCLUSIVE locks to be acquired on all databases. After a BEGIN EXCLUSIVE, you are guaranteed that no other thread or process will be able to read or write the database until the transaction is complete.

The default behavior for SQLite version 3.0.8 is a deferred transaction. For SQLite version 3.0.0 through 3.0.7, deferred is the only kind of transaction available. For SQLite version 2.8 and earlier, all transactions are exclusive.

The COMMIT command does not actually perform a commit until all pending SQL commands finish. Thus if two or more SELECT statements are in the middle of processing and a COMMIT is executed, the commit will not actually occur until all SELECT statements finish.

An attempt to execute COMMIT might result in an SQLITE_BUSY return code. This indicates that another thread or process had a read lock on the database that prevented the database from being updated. When COMMIT fails in this way, the transaction remains active and the COMMIT can be retried later after the reader has had a chance to clear.

## SQLite Pragma

PRAGMA database_list
PRAGMA foreign_key_list(table-name)
PRAGMA index_info(index-name)
PRAGMA index_list(table-name)
PRAGMA table_info(table-name)
PRAGMA [database.]schema_version
PRAGMA [database.]schema_version = integer
PRAGMA [database.]user_version
PRAGMA [database.]user_version = integer
PRAGMA journal_mode;
PRAGMA database.journal_mode;
PRAGMA journal_mode = DELETE | PERSIST | OFF
PRAGMA database.journal_mode = DELETE | PERSIST | OFF

# SQLite Notes

## 1. Storage Classes

Each value stored in an SQLite database has one of the following storage classes:

- **NULL**. The value is a NULL value.
- **INTEGER**. The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
- **REAL**. The value is a floating point value, stored as an 8-byte IEEE floating point number.
- **TEXT**. The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16-LE).
- **BLOB**. The value is a blob of data, stored exactly as it was input.

Any column except an INTEGER PRIMARY KEY may be used to store any type of value.  The exception to this rule is 'Strict Affinity Mode'.

Storage classes are initially assigned as follows:

Values specified as literals as part of SQL statements are assigned storage class TEXT if they are enclosed by single or double quotes, INTEGER if the literal is specified as an unquoted number with no decimal point or exponent, REAL if the literal is an unquoted number with a decimal point or exponent and NULL if the value is a NULL.  Literals with storage class BLOB are specified using the X'ABCD' notation.

Values supplied using the sqlite3_bind_* APIs are assigned the storage class that is requested.


## 2. Column Affinity

The type of a value is associated with the value itself, not with the column or variable in which the value is stored.  (This is sometimes called manifest typing.)  All other SQL databases engines that we are aware of use the more restrictive system of static typing where the type is associated with the container, not the value.

SQLite support the concept of "type affinity" on columns. The type affinity of a column is the recommended type for data stored in that column. The key here is that the type is recommended, not required.  Any column can still store any type of data, in theory.  It is just that some columns, given the choice, will prefer to use one storage class over another. The preferred storage class for a column is called its "affinity".

Each column in SQLite is assigned one of the following type affinities:

- **TEXT**
- **NUMERIC**
- **INTEGER**
- **NONE**

A column with TEXT affinity stores all data using storage classes NULL, TEXT or BLOB.  If numerical data is inserted into a column with TEXT affinity it is converted to text form before being stored.

A column with NUMERIC affinity may contain values using all five storage classes.  When text data is inserted into a NUMERIC column, an attempt is made to convert it to an integer or real number before it is stored.  If the conversion is successful, then the value is stored using the INTEGER or REAL storage class.  If the conversion cannot be performed the value is stored using the TEXT storage class.  No attempt is made to convert NULL or blob values.

A column that uses INTEGER affinity behaves in the same way as a column with NUMERIC affinity, except that if a real value with no floating point component (or text value that converts to such) is inserted it is converted to an integer and stored using the INTEGER storage class.

A column with affinity NONE does not prefer one storage class over another. It makes no attempt to coerce data before it is inserted.


## 3. Determination Of Column Affinity

The type affinity of a column is determined by the declared type of the column when the table is created, according to the following rules:

If the datatype contains the string "INT" then it is assigned INTEGER affinity.

If the datatype of the column contains any of the strings "CHAR", "CLOB", or "TEXT" then that column has TEXT affinity.  Notice that the type VARCHAR contains the string "CHAR" and is thus assigned TEXT affinity.

If the datatype for a column contains the string "BLOB" or if no datatype is specified then the column has affinity NONE.

Otherwise, the affinity is NUMERIC.


## 4. Row ID and the AUTOINCREMENT keyword

In SQLite, every row of every table has an 64-bit signed integer ROWID. The ROWID for each row is unique among all rows in the same table.

You can access the ROWID of an SQLite table using one the special column names ROWID, _ROWID_, or OID.  Except if you declare an ordinary table column to use one of those special names, then the use of that name will refer to the declared column not to the internal ROWID.

If a table contains a column of type INTEGER PRIMARY KEY, then that column becomes an alias for the ROWID.  You can then access the ROWID using any of four different names, the original three names described above or the name given to the INTEGER PRIMARY KEY column.  All these names are aliases for one another and work equally well in any context.

When a new row is inserted into an SQLite table, the ROWID can either be specified as part of the INSERT statement or it can be assigned automatically by the database engine.  To specify a ROWID manually, just include it in the list of values to be inserted.  For example:

CREATE TABLE test1(a INT, b TEXT);
INSERT INTO test1(rowid, a, b) VALUES(123, 5, 'hello');

If no ROWID is specified on the insert, an appropriate ROWID is created automatically.  The usual algorithm is to give the newly created row a ROWID that is one larger than the largest ROWID in the table prior to the insert. If the table is initially empty, then a ROWID of 1 is used. If the largest ROWID is equal to the largest possible integer (9223372036854775807) then the database engine starts picking candidate ROWIDs at random until it finds one that is not previously used.

The normal ROWID selection algorithm described above will generate monotonically increasing unique ROWIDs as long as you never use the maximum ROWID value and you never delete the entry in the table with the largest ROWID.  If you ever delete rows or if you ever create a row with    the maximum possible ROWID, then ROWIDs from previously deleted rows might be reused when creating new rows and newly created ROWIDs might not be in strictly accending order.

**The AUTOINCREMENT Keyword**

If a column has the type INTEGER PRIMARY KEY AUTOINCREMENT then a slightly different ROWID selection algorithm is used.  The ROWID chosen for the new row is at least one larger than the largest ROWID that has ever before existed in that same table.  If the table has never before contained any data, then a ROWID of 1 is used.  If the table has previously held a row with the largest

possible ROWID, then new INSERTs are not allowed and any attempt to insert a new row will fail with an SQLITE_FULL error.

SQLite keeps track of the largest ROWID that a table has ever held using the special SQLITE_SEQUENCE table. The SQLITE_SEQUENCE table is created and initialized automatically whenever a normal table that contains an AUTOINCREMENT column is created. The content of the SQLITE_SEQUENCE table can be modified using ordinary UPDATE, INSERT, and DELETE statements. But making modifications to this table will likely perturb the AUTOINCREMENT key generation algorithm. Make sure you know what you are doing before you undertake such changes.

The behavior implemented by the AUTOINCREMENT keyword is subtly different from the default behavior. With AUTOINCREMENT, rows with automatically selected ROWIDs are guaranteed to have ROWIDs that have never been used before by the same table in the same database. And the automatically generated ROWIDs are guaranteed to be monotonically increasing. These are important properties in certain applications. But if your application does not need these properties, you should probably stay with the default behavior since the use of AUTOINCREMENT requires additional work to be done as each row is inserted and thus causes INSERTs to run a little slower.

Note that "monotonically increasing" does not imply that the ROWID always increases by exactly one. One is the usual increment. However, if an insert fails due to (for example) a uniqueness constraint, the ROWID of the failed insertion attempt might not be reused on subsequent inserts, resulting in gaps in the ROWID sequence. AUTOINCREMENT guarantees that automatically chosen ROWIDs will be increasing but not that they will be sequential.

## SQLite Foreign Key

slExe "create table foo (id INTEGER NOT NULL PRIMARY KEY)

slexe "CREATE TABLE bar1 (id INTEGER NOT NULL PRIMARY KEY, foo_id INTEGER NOT NULL CONSTRAINT fk_foo_id REFERENCES foo(id) ON DELETE CASCADE)

slexe "CREATE TABLE bar2 (id INTEGER NOT NULL PRIMARY KEY, foo_id INTEGER NOT NULL, FOREIGN KEY (id) REFERENCES foo(id))

# SQLite Getting Schema Data

If you are running the sqlite3 command-line access program you can type ".tables" to get a list of all tables. Or you can type ".schema" to see the complete database schema including all tables and indices. Either of these commands can be followed by a LIKE pattern that will restrict the tables that are displayed.

From within a program you can get access to table and index names by doing a SELECT on a special table named "SQLITE_MASTER". Every SQLite database has an SQLITE_MASTER table that defines the schema for the database. The SQLITE_MASTER table looks like this:

```
CREATE TABLE sqlite_master (
type TEXT,
name TEXT,
tbl_name TEXT,
rootpage INTEGER,
sql TEXT
);
```

For tables, the type field will always be 'table' and the name field will be the name of the table. So to get a list of all tables in the database, use the following SELECT command:

```
SELECT name FROM sqlite_master
WHERE type='table'
ORDER BY name;
```

For indices, type is equal to 'index', name is the name of the index and tbl_name is the name of the table to which the index belongs. For both tables and indices, the sql field is the text of the original CREATE TABLE or CREATE INDEX statement that created the table or index. For automatically created indices (used to implement the PRIMARY KEY or UNIQUE constraints) the sql field is NULL.

The SQLITE_MASTER table is read-only. You cannot change this table using UPDATE, INSERT, or DELETE. The table is automatically updated by CREATE TABLE, CREATE INDEX, DROP TABLE, and DROP INDEX commands.

Temporary tables do not appear in the SQLITE_MASTER table. Temporary tables and their indices and triggers occur in another special table named SQLITE_TEMP_MASTER. SQLITE_TEMP_MASTER works just like SQLITE_MASTER except that it is only visible to the application that created the temporary tables. To get a list of all tables, both permanent and temporary, one can use a command similar to the following:

```
SELECT name FROM
```

```
(SELECT * FROM sqlite_master UNION ALL
SELECT * FROM sqlite_temp_master)
WHERE type='table'
ORDER BY name
```

# User Guide
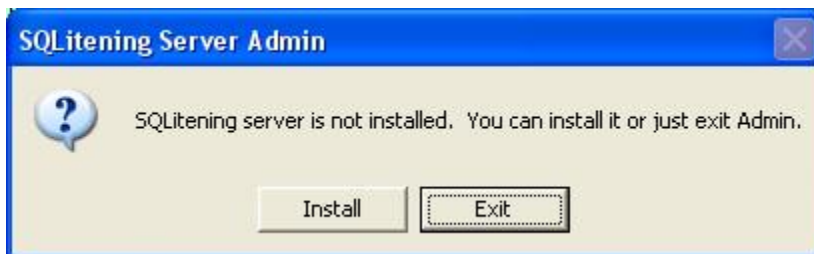
[Installation](#)

[Server Configuration](#)

[Server Log](#)
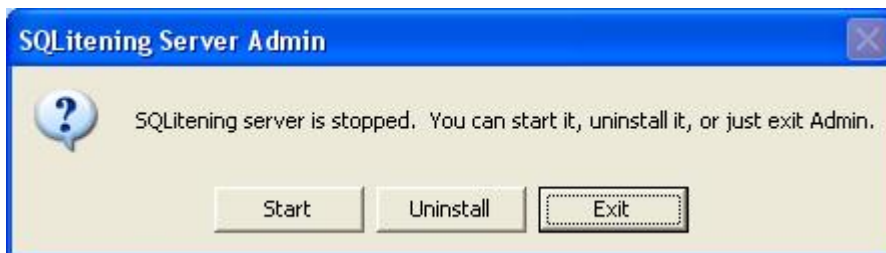
## Installation

It couldn't be easier...

- Download the project files if you have not done so yet.
- Copy all files from the zip file to a folder of your choice. Keep the folder structure intact.
- Double click the Admin program - SQLiteningServerAdmin.Exe.



- Click on install. You will be notified that the Service is installed.



- Click on OK.



- Now start the service by clicking on Start.

You are done! You can now try out the Examples.

# Server Configuration

SQLitening Server configuration takes place in the SQLiteningServer.Cfg file.
The file has the following sections:
[General]
[FACT]

A typical configuration would look like this:

[General]
ServiceNameSuffix =
Port = Default
Hosts = LocalHost, Beeblebrox, 192.168.178.29
LogConnDcon=Yes
LogInvalidInMessage=Yes
CreateDatabaseAllowed=Yes
TrimLogManually=No
MaxChunkSize=
ConnectionTimeOut=

[FACT]
Test.Sld = Goffy, AnotherPassWord,   ReadOnly%Password,
TestX.Sld = Goffy, AnotherPassWord,  ReadOnly%Password
>AEntry1 = pass
TestData\NewspaperNew.Jpg = *
Test1.Dat   =*
Test1X.Dat  =!

## Documentation of Sections and Entities

**[General]**
- **ServiceNameSuffix = suffix**  ---   defaults to none. Having a unique suffix will allow multiple services (servers) to be running on same computer.  The port numbers must also be different. Be sure to uninstall a service before changing.
- **Port = number**  ---   if omitted or value is empty or Default then will use 51234
- **Hosts = host1, host2, host3, ...** --- at least one host is required, can be LocalHost, a host name, or n.n.n.n
- **LogConnDcon = Yes or No**  ---  Controls the logging of connect and disconnect. If Yes then each connect and disconnect will be logged. If omitted then will default to No
- **LogInvalidInMessage = Yes or No**  ---   Controls the logging of invalid incoming messages.  If Yes and the message is invalid (first is not slConnect or is a wrong length) is received then an Error message is logged. If omitted then will default to No.   No will prevent hackers / attackers from filling the log.

- **CreateDatabaseAllowed = Yes or No** --- Controls the creation of new databases. If Yes then clients are allowed to create new databases on the server. If omitted then will default to No
- **TrimLogManually = Yes or No** --- Controls the trimming of the log when it becomes large (> 600K). If Yes then no automatic trimming will occur. If no then will automatically trim 100K from front of log when it becomes large. If omitted then will default to No.
- **MaxChunkSize** = Number K --- Controls the size of Row Data Chunks which are returned from Select statements. The value is in K so the actual size is * 1000. Default is 500.
- **ConnectionTimeOut** = Number Minutes --- Control the number of minutes the server will wait to receive a message from an active connection. Default is 30. Set to -1 if you want to never timeout.


## [FACT] (File Access Control Table)
- **filename = password, password, ..**.

## Details:
- Password of * will match any password(same as removing file.
  Password of ! will refuse access to all, protected file.
- Password of blank will match a blank password or no password.
- Password containing one or more percent sign(%) characters will require the file to be opened as read-only. Attaching a file with a read-only password will fail if the slOpen was not read-only.To password protect Procs, use the proc name as a filename but preceed it with a > so it will not duplicate a real file name.

## Note:
slPutFile requires the file name be in the FACT.

## Server Log

### SQLitening Server Log

This an excerpt of a server log:

```
090108100639 Admn Strt ==========<[ Start Server ]>==========
090108100639 Admn Flgs TrimLogManually=No
090108100639 Admn Flgs LogConnDcon=Yes
090108100639 Admn Flgs LogInvalidInMessage=Yes
090108100639 Admn Flgs CreateDatabaseAllowed=Yes
090108100639 Admn FACT Loaded
090108100640 Host LocalHost(127.0.0.1) Socket=356 Port=51234
090108100640 Host 192.168.178.25(192.168.178.25) Socket=376 Port=51234
090108100640 Host 192.168.178.27(192.168.178.27) Socket=372 Port=51234
090109084514 Conn #5 SK 376 rbsoft192.168.178.29
090109084805 Dcon #5
090109093045 Conn #9 SK 376 rbsoft192.168.178.29
090109093221 Dcon #9
090109093742 Conn #16 SK 376 rbsoft192.168.178.29
090109094006 Dcon #16
090109095223 Conn #23 SK 372 rbsoft192.168.178.29
090109095305 Dcon #23
090109193851 Conn #36 SK 376 rbsoft192.168.178.29
090109193856 Dcon #36
090109193917 Conn #40 SK 376 rbsoft192.168.178.29
090109194701 Dcon #40
090113122903 Admn Stop ==========<[ Stop Server ]>==========
```

**The layout is as follows:**
  **1 -  6 =**    **Date** as YYMMDD
  **7 - 12 =**    **Time** as HHMMSS
 **14 - 17 =**    **Entry Type** as:
          **Admn** -- followed by a 4 byte message type of Strt, Stop, Flgs, Data, Info, or **FACT** and then the message.
          **Host** -- followed by the host name, IP address, socket, port, and service name suffix.
          **Conn** -- followed by # and the TCP file number and "SK" and the socket number and user data.
          User data consists of user name, computes name, and IP address delimited by $BS. Any part of the user data may be omitted by the calling program.
          **Dcon** -- followed by # and the TCP file number and one of the following: "Dropped" "Disconnect" "TimeOut" "Killed" "Error"
          **Eror** -- followed by the error message.
          **Exit** -- followed by the custom message from

SQLiteningServerExit.Dll

**19 - ??** =     **Data** depending upon Record Type above.

The TCP file numbers in Conn, User, and Dcon allow you to match up connects with disconnects and compute elapsed time for a user.

## Misc

[Credits](#)

[Links](#)

## Credits

- The SQLitening Project is developed and maintained by Fred Meier.
- The project is hosted by Planet Squires Software..
- This help file is maintained by Rolf Brandt.

The help file was created with the free HelpMaker software.

# Links

**Home of the SQLitening Database System, latest news, download, and Forum:**
[http://www.sqlitening.com/support/index.php](http://www.sqlitening.com/support/index.php)