LiteX

LiteX package consists of:

- **LiteX Automation** – automation wrapper around SQLite3 library.
  This wrapper is useful if you want to use SQLite3 databases in Visual Basic or any scripting languages such as JScript,VBScript and (my favorite) AutoIt3.
- **LiteX++** - C++ wrapper around SQLite3 library.
  This wrapper is useful to build fast SQLite3 database access in your C++ projects.
- **LiteX ADO .NET** provider - beta code.
  ADO .NET provider for SQLite3 databases to use SQLite3 databases in managed code.

All these wrappers comes with full source and are public domain.

Homepage: http://www.assembla.com/wiki/show/litex.
Public (read-only) SVN repository: http://svn2.assembla.com/svn/aIPblab5qr3zkdab7jnrAJ.

Author: Edmunt Pienkowsky.
E-Mail: roed@onet.eu.

LiteX Automation.

LiteX Automation is an automation wrapper over SQLite3 library.

This library is build in two versions: `sqlite3.dll` and `sqlite3u.dll`. Each of them exports all known `sqlite3_...` functions and few new functions listed below. `sqlite3.dll` uses UTF-8 text encoding and `sqlite3u.dll` uses UTF-16LE text encoding and it uses Unicode runtime library. Use Unicode version of LiteX Automation on NT-bases OS'es (WinNT, Win2K, WinXP, Win2003, etc.). Automation objects may be registered using **`regsvr32`** tool:

```
regsvr32 sqlite3.dll
```

You may unregister automation objects by typing:

```
regsvr32 /u sqlite3.dll
```

Please register this library every time you download new version because interface definition may be changed.

If you have troubles with `sqlite3.dll` library registration it means that you probably have another `sqlite3.dll` library into your system directory. Putting `sqlite3.dll` into system directory is a **very bad idea** so make sure that this library in this location is really necessary. If you **really** need this library into system directory please copy LiteX Automation library into system directory and register it. In most situations this solution works but version compatibility problems may occurs – be careful. Another solution is to run `regsvr32` tool as follows:

```
regsvr32 .\sqlite3.dll
```

or specify full path to this library even if you run `regsvr32` from directory where `sqlite3.dll` resides.

Automation objects.

Connection.

```
oDb = new ActiveXObject("LiteX.LiteConnection")
```

Connection object calls `sqlite3_register_blob_functions` and `sqlite3_register_unacc_functions` so new functions and collation sequences may be used without extra work.

Properties.

Property	Access mode	Description	Examples
`Version( [string= TRUE] )`	**RO**	String that describes SQLite engine version. • `string = TRUE` (default) String is returned. For example `3.4.2` • `string = FALSE` Numerical value is returned. For example `34020`.	`WScript.Echo( "SQLite3 version", oDb.Version, oDb.Version(false) );`
`Path`	**RO** - if database file is open **RW** - if database file isn't open	Path to database file.	`oDb.Path = "c:\\temp\\db.db"`
`Changes`	**RO** - if database file is open **NA**- if database file isn't open	The number of database rows that were changed (or inserted or deleted) by the most recent database operation.	`WScript.Echo( "Last changes", oDb.Changes );`
`LastInsertRowid`	**RO** - if database file is open **NA**- if database file isn't open	Integer key of the most recent insert in the database.	`WScript.Echo( "Last ID", oDb.LastInsertRowid );`
`ProgressPeriod`	**RW**	This property determines frequency of generating `Progress` event. Default value is `0` - `Progress` event will be never raised. If value of this property is greater than zero then `Progress` event will be raised every `ProgresPeriod` sqlite engine virtual opcode. See documentation of natiive `sqlite3_progress_handler` (second parameter) function for more information.	`oDb.ProgressPeriod=100`

Methods.

Method	Description	Examples

`Open( [path] )`	Opens selected database. Parameter `path` may be omitted if you set `Path` property before.	`oDb.Open();`
		
		If you use Unicode versionn of LiteX Automation remember that newly created database is UTF-16LE encoded by default. If you need UTF-8 encoded database Use `PRAGMA encoding` statement to set default encoding of newly created database.
		
		`oDb.BatchExecute("PRAGMA encoding='UTF-8'");`

`OpenInMemory()`	Opens in-memory database.
	Contents of in-memory database is destroyed when database is closed.
		Code:
		
		`oDb.OpenInMemory();`
		
		is equivalent to
		
		`oDb.Open( ":memory:" );`

`Execute( sql, ... )`	Fast way to execute single query `sql`. This query may include parameters. Parameters values you specyfy as additional method parameters - see examples.
	If query doesn't returns any result empty value is returned.
	If query returns one or more rows first row is returned as `Row` property of statement object does.
		`oDb.Execute( "INSERT INTO Test(a,b) VALUES (?,?)", "foo", 12.45 );`
		`oDb.Execute( "INSERT INTO Test(a,b) VALUES (?,?)", null, "peacemaker" );`
		`nMax = oDb.Execute( "SELECT max(a) FROM Test" );`

`BatchExecute( sql )`	Executes many SQL statements at once. Statements should be non-query but this is not necessary.	`oDb.BatchExecute( "CREATE TABLE Test(a); CREATE INDEX idx ON Test(a); CREATE TestAgain(a); CREATE INDEX idx_again ON TestAgain(a)" )`

Method	Description	Examples
`Prepare( sql )`	Prepares SQL statement and returns prepared statement object.	`oStmt = oDb.Prepare( "SELECT a,b FROM Test WHERE a > 10" );`
`Close()`	Closes database. You must close all statements associated with this database before.	`oDb.Close();`
`Interrupt()`	Interrupts execution of pending query. You may use this method into event handler.	`oDb.Interrupt();`

Events.

Event	Interface	Description
`Progrss( ByRef abort )`	**`IliteProgress (default)`**	Progress event is generating during execution of long query and if `ProgressPeriod` property is set to value greater than zero. Last parameter of this event is a boolean flag **specified by reference**. Setting this flag to FALSE allow you to abort pending operation. Please note that in many languages you cannot pass value back from event handler. In such situation you can use `Interrupt` method of connection object.
`Busy( counter, ByRef abort )`	`ILiteBusy`	Busy event is generated when sqlite engine is trying to access database file and this file is locked. The `counter` parameter indicates how many this event was raised. Last parameter of this event is a boolean flag s**pecified by reference**. Setting this flag to TRUE sqlite engine try access database file again. By setting this flag to FALSE sqlite engine stops trying to access database file and generates error. Please note that in many languages you cannot pass value back from event handler.

Statement.

Statement object typically is created by `Prepare` method of connection object but you may create it manually:

```
oStmt = new ActiveXObject("LiteX.LiteStatement");
```

Properties.

Property	Access mode	Description	Examples
`ActiveConnection`	**RW** - if not prepared **RO** - if prepared	Connection object associated with this statement.	`oStmt.ActiveConnection = oDb;`
`CommandText`	**RW** - if not prepared **RO** - if prepared	Statement text - SQL query.	`oStmt.CommandText = "SELECT a,b FROM Test ORDER BY a DESC";`
`ColumnCount`	**NA** - if not prepared **RO**- if prepared	Number of columns returned by statement.	
`ColumnName( idx )`	**NA** - if not prepared **RO**- if prepared	Name of `idx` column. Columns are numbered from `0` to `ColumnCount-1`.	
`ColumnType( idx )`	**NA** - if not prepared **RO**- if prepared	Type of `idx` column; `idx` may be: • integer – column index • string – column name Return values: • `lxNull` = 0 – NULL value • `lxInteger=1` – integer value • `lxLongInteger=2` – integer value • `lxFloat=3` – floating point value • `lxString=4` – string • `lxBinary=5` – binary (BLOB)	

Property	Access mode	Description	Examples
`ColumnValue( idx, [type=lxUnknown] )`	**NA** - if not prepared **RO**- if prepared	Value of `idx` column; `idx` may be: • integer – column index • string – column name You may force return type by optional `type` parameter . Possible values of `type`: • `lxUnknown=-1` – guess column type, default • `lxInteger` – integer value • `lxLongInteger` – integer value • `lxFloat` – floating point value • `lxString` – text • `lxBinary` – binary value (BLOB) • `lxDate` – DATE type, note that date is stored as float number so date type cannot be guessed, if you need DATE value you must specify type parameter	
`Row( [mode=lxDefault] )`	**NA** - if not prepared **RO**- if prepared	• `mode=lxDefault` (default value) - returns whole row in one dimension array. • `ColumnCount` = 0 – empty value is returned. • `ColumnCount` = 1 – no array is returned but one single scalar value. • `mode=lxArray` – always one-dimension array is returned. • `mode=lxCollection` – `Row` collection is returned. This is default property.	`row = oStmt.Row;`
`ParameterCount`	**NA** - if not prepared **RO**- if prepared	Number of statement's parameters to bind.	
`ParameterName( idx )`	**NA** - if not prepared **RO**- if prepared	Name of `idx`-th parameter. Parameters are numbered from `1` to `ParameterCount`. If parameter is nameless or `idx` is out of range empty string is returned.	
`Done`	**NA** - if not prepared **RO**- if prepared	Indicates that `Step` method returns without any results, this may be end of record set for example.	

`RowCount`	**NA** - if not prepared
	RO- if prepared
		Returns number of rows returned by statement.	Before use this property you should understand how this property works. Code:
			
			`nCount = oStmt.RowCount`
			
			is equivalent to:
			
			`nCount = 0;`
			`oStmt.Reset();`
			`while ( !oStmt.Step() ) nCount++;`
			`oStmt.Reset();`
			
			As you see to determine number of rows all rows must be iterated. **This may take long time.** `RowCount` property is a little bit faster because it doesn't use expensive Automation calls. Use this property with caution and inside transaction.

Collections

Collection (Property)	Access mode	Description	Examples

`Rows( [static=FALSE], [maxrec=0] )`	**NA** - if not prepared
	RO- if prepared
		Returns collection of `Row` objects. This is a `Rows` object.
		• `static=TRUE` – returns static collection. All records will be iterated and results will be stored in memory. You can limit number of records by `maxrec` parameter (zero by default – no limit).
		• `static=FALSE` (default) – returns dynamic collection.
			`For Each oRow In oStmt.Rows`
			`  ...`
			`Next`

Collection (Property)	Access mode	Description	Examples

`Columns`	**NA** - if not prepared
	RO- if prepared
		Returns collection of `Column` objects. This is a `Columns` object.	`For Each oColumn In oStmt.Columns`
			`...`
			`Next`

`Parameters`	**NA** - if not prepared
	RO- if prepared
		Returns collection of `Parameter` objects. This is a `Parameters` object.	`For Each oParameter In oStmt.Parameters`
			`...`
			`Next`

Methods.

Method	Description	Examples
`Prepare( [sql] )`	Prepares statement, parameter `sql` may be omitted if you set `CommandText` property before.	`oStmt.Prepare();`

`BindParameter( idx, [value], [type=lxUnknown] )`	Binds value to parameter; `idx` may be:
	• integer - parameter index, parameters are numbered from **one** to `ParameterCount`
	• string - parameter name, only if named parameters are used in SQL command
	
	Possible values of `type`:
	• `lxUnknown=-1` – guess value type, default
	• `lxNull` – bind NULL value, `value` is ignored
	• `lxInteger` – bind integer value
	• `lxLongInteger` – bind integer value
	• `lxFloat` – bind floating point value
	• `lxString` – bind text
	• `lxBinary` – bind binary value (BLOB)
	• `lxDate` – date type (DATE)
		`oStmt.BindParameter( 1, null );`
		`oStmt.BindParameter( 1, "123", 1 /*lxInteger*/ );`

`BindParameters(...)`	One-call parameters binding. Instead of calling `BindParameter` method many times you may call BindParameters at once.	`oStmt.BindParameters( 1, null, "Hello", 1.2222 )`

`Step( [steps=1] )`	Makes `steps` steps of statement execution. Returns `Done_` property value.
	Statement must be prepared before use this method. Default value of nSteps is one - one step (next row).
		`while( !oStmt.Step() );`
		`oStmt.Step(10);`

`Execute()`	Non-query statement execution.
	After execution statement is ready to re-execute.
		`oStmt.Prepare( "INSERT INTO Table(a) VALUES (?)");`
		`for( i=0; i<100; i++ )`
		`{`
		`  oStmt.BindParameter( 1, i );`
		`  oStmt.Execute();`
		`}`
		
		Instead of calling `Execute()` method you can use following sequence:
		
		`oStmt.Step();`
		`oStmt.Reset();`

`Reset()`	Resets statement. Begins execution of statement. Parameters binding remains but may be changed.	`oStmt.Reset();`
`Close()`	Closes statement.	`oStmt.Close();`

Rows.

```
LiteX.LiteRows
```

You cannot create Rows collection directly. It is returned by `Rows` property of statement object. This is collection of `Row` objects.

Property	Description	Examples
`Count`	Number of elements. Only if collection is static.	`nCount = oRows.Count`

`Item( [idx=-1] )`	Returns row of specified zero-based index.
	If collection is non-static then `.idx` parameter is ignored and current row is returned.
	This is default property.
		`for( i=0; i < nCount; i++ )`
		`{`
		`  oRow = oRows(i)`
		`}`

Row.

```
LiteX.LiteRow
```

You cannot create Row collection directly. `Rows` collection contains elements of this type. Row itself is collection of values.

Property	Description	Examples
`Item( idx, [type=lxUnknown] )`	Returns value of `idx`-th column. Specifying second parameter you can force return value type. This is default property.	`val = oRow(0)`
`Count`	Returns numbers of columns.	`columns = oRow.Count`
`Value`	Returns all values of all columns in one-dimension array.	

Columns.

```
LiteX.LiteColumns
```

You cannot create Columns collection directly. It is returned by `Columns` property of statement object. This is collection of `Column` objects.

Property	Description	Examples
`Count`	Number of columns.	`nCount = oColumns.Count`

`Item( idx )`	Returns column of specified index idx.
	This is default property.
	
	`idx` may be:
	• integer – column zero-based index
	• string – column name
		`for( i=0; i < nCount; i++ )`
		`{`
		`  oColumn = oColumns(i)`
		`}`

Column.

```
LiteX.LiteColumn
```

You cannot create this object directly. `Columns` collection contains elements of this type.

Property	Description	Examples
`Index`	Returns index of column.	

`Value`	Returns name of column.
	This is default property.
		`For Each oColumn In oStmt.Columns`
		`  Wscript.Echo "Column index:", oColumn.Index`
		`  Wscript.Echo "Column name:",oColumn`
		`Next`

Parameters.

```
LiteX.LiteParameters
```

You cannot create Parameters collection directly. It is returned by `Parameters` property of statement object. This is collection of `Parameter` objects.

Properties.

Property	Description	Examples
`Item( idx )`	Returns `idx`-th parameter object. This is default property. `idx` may be: • integer – column zero-based index • string – column name	`oParam = oStmt.Parameters(0)`
`Count`	Returns number of parameters in collection.	

Methods.

Method	Description	Examples

`Bind( ... )`	Binds values to parameters at once. This method do the same such `BindParameters` method of statement object.	`oStmt.Parameters.Bind( 1,2,3 )`
		is equvalent to
		`oStmt.BindParamaters( 1,2,3 )`

Parameter.

```
LiteX.LiteParameter
```

You cannot create this object directly. `Parameters` collection contains elements of this type.

Properties.

Property	Description	Examples
`Index`	Index of property.	
`Name`	Name of property. This is default property.	

Methods.

Method	Description	Examples

`Bind( value, [type=lxUnknown] )`	Bind specified `value` to parameter. You may force type of binded value.	`oStmt.Parameters(0).Bind( "123" )`
		is equvalent to
		`oStmt.BindParameter(0,"123")`

LargeInteger.

This class is a helper class. It represents 64-bit integer. It may help you to work with 64-bit integers.

```
oLi = new ActiveXObject("LiteX.LargeInteger");
```

Property	Access mode	Description	Examples
`LowPart`	**RW**	Low 32 bits of large integer.	`oLi.LowPart = 0xffffffff`
`HighPart`	**RW**	High 32 bits of large integer.	`oLi.HighPart = 0xffffffff`
`QuadPart`	**RW**	Value of large integer as DECIMAL. This is default property.	`oLi.QuadPart = "9999999999999999"`
`QuadPartCy`	**RW**	Value of large integer as CY.	`oLi.QuadPartCy = CCur(999999999999.9999)`
`MIN_VALUE`	**RO**	Minimum available value as DECIMAL.	`WScript.Echo oLi.MIN_VALUE`
`MAX_VALUE`	**RO**	Maximum available value as DECIMAL.	`WScript.Echo oLi.MAX_VALUE`
`MIN_VALUE_CY`	**RO**	Minimum available value as CY.	`WScript.Echo oLi.MIN_VALUE_CY`
`MIN_VALUE_CY`	**RO**	Maximum available value as CY.	`WScript.Echo oLi.MAX_VALUE_CY`

Additional functions.

sqlite3_register_blob_functions.

```
void sqlite3_register_blob_functions(sqlite3*)
```

This function register two new functions for sqlite3 engine that works on blob fields:

Function	Description	Example

`tovis( blob <,repl> )`	Converts `blob` argument to string. Unprintable characters are converted to `repl` character (default `"."`).	`SELECT tovis( b ) FROM blober;`
		`SELECT tovis( b, "_") FROM blober;`

`tohex( blob <,sep> )`	Converts `blob` argument to string with hexadecimal notation. Optionally bytes may be separated by `sep` character (no separator by default).	`SELECT tohex( b ) FROM blober;`
		`SELECT tohex( b, "|") FROM blober;`

sqlite3_register_unacc_functions.

```
void sqlite3_register_unacc_functions(sqlite3*)
```

This function register one function for sqlite3 engine that works on text fields:

Function	Description	Example
`unaccent( txt )`	Removes accents from specified text.	`SELECT unaccent( b ) FROM some_table;`

Additionally it registers two collation sequences:

Collation sequence	Description	Example
`unaccented`	Compares unaccented text. Case sensitive version.	`CREATE Table( t TEXT COLLATION unaccented );`
`unaccentedi`	Compares unaccented text. Case insensitive version.	`CREATE Table( ti TEXT COLLATION unaccentedi );`

This function returns pointer which must be used in `sqlite3_unregister_unacc_functions` as second parameter.

sqlite3_unregister_unacc_functions.

```
void sqlite3_unregister_unacc_functions( sqlite3*, void*)
```

This function unregisters sqlite3 functions registered previously by `sqlite3_register_unacc_functions`. The second parameter is a pointer returned by `sqlite3_register_unacc_functions` function.

Handling 64-bit integers.

Handling 64-bit integers in Automation is rather complicated due to some limitations of Automation itself and some limitation of languages that using Automation objects. To return 64-bit integers LiteX Automation uses DECIMAL structure. DECIMAL is Automation-compatible but many languages doesn't support this type properly. Visual Basic and VBScipt knows this type but you cannot perform any arithmetical operation on this type. In JScript all numbers stored in DECIMAL are automatically converted to float (Numerical) type. If you need to perform some arithmetical operations on 64-bit integers you may use CY (currency) type. Internally CY is stored as 64-bit integer. By using `LargeInteger` class and its `QuadPartCy` property you can use CY value as 64-bit integer:

```
oLi.QuadPart = oDb.LastInsertRowid
```

```
WScript.Echo "Current ROWID:", oLi.QuadPart
oLi.QuadPartCy = oLi.QuadPartCy + CCur(0.0001)  ' +1
WScript.Echo "Next ROWID:", oLi.QuadPart
```

Seems complicated? Yes it is but this is not LiteX limitation.

Binding 64-bit integers.

To bind 64-bit integer you must force type to `lxInteger` (1):

```
oStmt.Paramaeters(":largeint").Bind( "1234567891011", 1 ) ; binding string
oStmt.Paramaeters(":largeint").Bind( oLi, 1 ) ; binding large integer object
```

Reading 64-bit values.

64-bit integers are returned as DECIMAL. But 32-bit integers are returned as Long.

```
oLi.QuadPart = oStmt.Row("largeint")
Wscript.Echo "Hi:", oLi.HighPart, "Lo:", oLi.LowPart
```

The example above doesn't work properly in JScript because DECIAML is internally converted to float (Numerical) type.

You may also force returned value type to string.

```
sLi = oStmt.ColumnValue( "largeint", 4 ) ;string returned
```

Access mode abbreviations.

Abbreviation	Meaning
RO	read only
RW	read and write
NA	not accessible, any access to this property generates error

Error codes.

Error code (hex)	Description	SQLite native error code
00000000	Successful result. Not an error.	`SQLITE_OK`
C0000001	SQL error or missing database.	`SQLITE_ERROR`
C0000002	An internal logic error in SQLite.	`SQLITE_INTERNAL`
C0000003	Access permission denied.	`SQLITE_PERM`
C0000004	Callback routine requested an abort.	`SQLITE_ABORT`
C0000005	The database file is locked.	`SQLITE_BUSY`
C0000006	A table in the database is locked.	`SQLITE_LOCKED`
C0000007	A malloc() failed – out of memory.	`SQLITE_NOMEM`
C0000008	Attempt to write a readonly database.	`SQLITE_READONLY`
C0000009	Operation terminated by `sqlite3_interrupt()`.	`SQLITE_INTERRUPT`
C000000A	Some kind of disk I/O error occurred.	`SQLITE_IOERR`
C000000B	The database disk image is malformed.	`SQLITE_CORRUPT`
C000000C	(Internal Only) Table or record not found.	`SQLITE_NOTFOUND`
C000000D	Insertion failed because database is full.	`SQLITE_FULL`

Error code (hex)	Description	SQLite native error code
C000000E	Unable to open the database file.	`SQLITE_CANTOPEN`
C000000F	Database lock protocol error.	`SQLITE_PROTOCOL`
C0000010	Database is empty.	`SQLITE_EMPTY`
C0000011	The database schema changed.	`SQLITE_SCHEMA`
C0000012	Too much data for one row of a table.	`SQLITE_TOOBIG`
C0000013	Abort due to constraint violation.	`SQLITE_CONSTRAINT`
C0000014	Data type mismatch.	`SQLITE_MISMATCH`
C0000015	Library used incorrectly.	`SQLITE_MISUSE`
C0000016	Uses OS features not supported on host.	`SQLITE_NOLFS`
C0000017	Authorization denied.	`SQLITE_AUTH`
C0000018	Auxiliary database format error.	`SQLITE_FORMAT`
C0000019	2nd parameter to BindParameter out of range.	`SQLITE_RANGE`
C000001A	File opened that is not a database file.	`SQLITE_NOTADB`
40000064	Another row ready. Not an error.	`SQLITE_ROW`
40000065	Finished query execution. Not an error.	`SQLITE_DONE`
C00000C8	Statement already prepared.	LiteX specific
C00000C9	Connection property not set.	LiteX specific
C00000CA	No SQL statement was given.	LiteX specific
C00000CB	Statement not prepared.	LiteX specific
C00000CC	Unknown binary data.	LiteX specific
C00000CD	Cannot guess data type.	LiteX specific
C00000CE	Cannot get column name.	LiteX specific
C00000CF	Unknown column type or bad column index.	LiteX specific
C00000D0	Cannot create statement object.	LiteX specific
C00000D1	Column index out of range.	LiteX specific
C00000D2	Unknown column name.	LiteX specific
C00000D3	Unknown column index type. Only string or integer values are allowed.	LiteX specific
C00000D4	Parameter index out of range.	LiteX specific
C00000D5	Unknown parameter name.	LiteX specific
C00000D6	Unknown parameter index type. Only string or integer values are allowed.	LiteX specific
C00000D7	Database file is open.	LiteX specific
C00000D8	Database file isn't open.	LiteX specific
C00000D9	Bad step parameter.	LiteX specific
C00000DA	Non-query statement returns row.	LiteX specific
C00000DB	Unsupported value type type.	LiteX specific
C00000DC	Bad binding value.	LiteX specific

Building LiteX Automation from source.

LiteX uses ATL library. The minimum required ATL version is 3.0.

To build LiteX binaries I'm using *Visual Studio 2005* compiler. For long time *Visual Studio 6.0* was used but for some reason I cannot install this application on my new computer. Project files (dsp,dsw) from VC++ 6.0 are still included but they may be **out of date** - modifications are simple but I cannot make them. If you have VC++ 6.0 compiler and want to help develop LiteX please contact me. For the same reason I cannot recompile VB example.

It is possible to compile LiteX using *Visual Studio C++ 2005 Express Edition*. If you use this free compiler you must install also latest *Platform SDK* and hack some ATL headers. See here for more details.

LiteX is by default compiled using my *libunacc* library. You can ommit this stuff using *"... no Unacc"* (e.g. *"Release no Unacc"*) configuration. Please specify *"... no Unacc"* configuration if during compilation `unacc.h` header (from libunacc library) is missing.

If you have problems with LiteX sources you can always contact me. I consider putting LiteX sources in some public repository. If you can help (where?, how?) please contact me too.

LiteX++

LiteX++ is a simple C++ wrapper around SQLite3 C native API. Sources of this library you can find in `LiteX_pp` subdirectory of LiteX package (see also library usage). To use this library you must have basic knowledge about C++ language and SQLite3 native C API.

Main features of LiteX++ library.

- **Most of classes methods are inline.**
  Because most of class methods are inline compiler can generate really fast code.
- **Support for Unicode (UTF-16LE).**
  LiteX++ uses `_T( )` macro and `TCHAR` psedo type from `tchar.h` header.
  In Unicode version of this library UTF-16 version of SQLite3 C API routines (eg. `sqlite3_open16`, `sqlite3_errmsg16)` are used whenever possible.
  In non-Unicode (ANSI) version of this library every string is encoded to UTF-8 string. This behavior makes that strings in SQLite3 database are stored as UTF-8 or UTF-16 text that makes SQLite3 database more portable. For example database created by this library can be easy accessed by LiteX Automation library and vice versa.
- **LiteX++ uses STL standard library.**
  STL library is mainly used to string handling. LiteX++ `typedef`-s own string type `_tstring` as `std::string_base<TCHAR>`.
  LiteX++ also throws exceptions derived from `std::runtime_class`.
- **Public domain code.**
  You may use this library whenever you want without any restrictions!

Class reference.

All classes and helper functions are groupped into `litex` namespace:

```
using namespace litex;
```

SQLiteException class.

This class is used by LiteX++ library to throw exceptions indicating error from SQLite3 library.

Methods.

Method(s)	Description	Sample code / Comments

`int get_ErrorCode() const`	Gets error code from SQLite3 library.
	Look at sqlite3.h header to see error codes and their descriptions.
		`try`
		`{`
		`   ....`
		`}`
		`catch( SQLiteException& e )`
		`{`
		`   tcerr << _T("Error code: ") << e.get_ErrorCode()`
		`<< endl;`
		`}`

`const _tstring& get_Message()`
`const`
	Gets error mesage from SQLite3 library.
	This is error text returned by `sqlite3_errmsg(16)` function.
		`try`
		`{`
		`   ....`
		`}`
		`catch( SQLiteException& e )`
		`{`
		`   tcerr << _T("Error message: ") << e.get_Message()`
		`<< endl;`
		`}`

`static void Throw( int`
`nErrorCode, sqlite3* pDb )`
`static void Throw( int`
`nErrorCode )`
	Throws SQLiteException when error code is not equal to SQLITE_OK.
	This methods are used internally by LiteX++ to throw SQLiteException exceptions when nessesary.
		`SQLiteConnection db;`
		`...`
		`SQLiteException::Throw( nErrorCode, db );`

SQLiteRuntimeException class.

This class is used by LiteX++ library to indicate its own runtime errors. Note that not all errors are indicated. For example parameters validation. Parameters validation is prformed only in DEBUG mode by `assert` macro/function from `<cassert>` header. This enables to produce fast code without unnecessary validation in RELEASE mode.

Methods.

Method(s)	Description	Sample code / Comments

`_tstring get_Message() const`	Gets error message.	`try`
		`{`
		`   ...`
		`}`
		`catch( SQLiteRuntimeException& e )`
		`{`
		`   tcerr << _T("LiteX++ exception: ") <<`
		`e.get_Message() << endl;`
		`}`

`static void Throw( const`
`_tstring& sMsg )`
	Throws SQLiteRuntimeException with specified error message.	

SQLiteConnection class.

This is a wrapper class around `sqlite3*` handle and represents connection to SQLite database.

Constructors.

Constructor	Description	Sample code / Comments

`SQLiteConnection()`	Default constructor.
	Object initialization will be performed in future.
		`SQLiteConnection db;`
		`db.Open( _T("some.db") );`
		`if ( db )`
		`{`
		`   db.Close();`
		`}`

`SQLiteConnection( const`
`_tstring& sDbPath )`
	Initializes object and opens sDbPath database file.
	If database file cannot be open SQLiteException exception is thrown.	

Constructor	Description	Sample code / Comments
`SQLiteConnection( const TCHAR* pszDbPath )`	Initializes object and opens pszDbPath database file. If database file cannot be open SQLiteException exception is thrown.	`SQLiteConnection db( _T("some.db") );`

`SQLiteConnection( SQLiteConnection& db )`	Copy constructor. Constructing object takes ownership of `sqlite3*` handle and `db` object is detached from this handle.	`SQLiteConnection db( _T("some.db") );`
		`if ( db ) tcout << _T("Database is opened.") << endl;`
		`SQLiteConnection other_db( db );`
		`if ( !db ) tcout << _T("Database is detached.") << endl;`
		`if ( other_db ) tcout << _T("Databasse is attached.") << endl;`

Methods.

Method(s)	Description	Sample code / Comments
`bool Open( const _tstring& sDbPath )`	Creates and/or opens `sDbPath` database file. Returns true if database is created/opened. In case of failure no exception is thrown and method return false.	Wrapper around `sqlite3_open(16)` function.
`bool OpenInMemory()`	Creates and opens empty in-memory database. Contents of in-memory database is destoryed when database is closed.	If you want to create in-memory database in constructor use MEMORY_DB string: `SQLiteConnection db( MEMORY_DB );`
`void Close()`	Closes previously opened database.	Wrapper around `sqlite3_close` function.
`void Interrupt()`	This function causes any pending database operation to abort and return at its earliest opportunity.	Wrapper around `sqlite3_interrupt` function.
`sqlite_int64 get_LastInsertRowid() const`	The following routine returns the integer key of the most recent insert in the database.	Wrapper around `sqlite3_last_insert_rowid` function.
`int get_Changes() const`	This function returns the number of database rows that were changed (or inserted or deleted) by the most recent executed statement.	Wrapper around `sqlite3_changes` function.
`int get_TotalChanges() const`	This function returns the number of database rows that have been modified by INSERT, UPDATE or DELETE statements since the database handle was opened.	Wrapper around `sqlite3_total_changes` function.

`void BatchExecute( const TCHAR* pszSql )`
`void BatchExecute( const _tstring& sSql )`
	Executes many SQL statements at once.	`db.BatchExecute( _T("CREATE INDEX d ON Test( d DESC ); CREATE INDEX e ON Test( e )") );`

`void ExecuteNonQuery( const TCHAR* szSql )`
`void ExecuteNonQuery( const _tstring& sSql )`
	Executes single SQL statement that doesn't returns any results.	`db.ExecuteNonQuery( _T("CREATE TABLE Test( a INTEGER PRIMARY KEY, b, c, d, e, f )") );`

Method(s)	Description	Sample code / Comments
`int ExecuteScalarInt( const _tstring& sSql )` `sqlite_int64 ExecuteScalarInt64( const _tstring& sSql )` `double ExecuteScalarDouble( const _tstring& sSql )` `_tstring ExecuteScalarText( const _tstring& sSql )` `int ExecuteScalarInt( const TCHAR* pszSql )` `sqlite_int64 ExecuteScalarInt64( const TCHAR* pszSql )` `double ExecuteScalarDouble( const TCHAR* pszSql )` `_tstring ExecuteScalarText( const TCHAR* pszSql );`	Executes singleton SQL statement (statement that returs one row with one column) and returns its result. If statement doesn't returns any row. SQLiteRuntimeException is thrown.	In fact statements may returs many rows witch many columns but only first row is fetched and and value from first column is returned. `int nMax = db.ExecuteScalarInt( _T("SELECT max(a) FROM test") );`
`SQLiteStatement Prepare( const _tstring& sSql)` `SQLiteStatement Prepare( const TCHAR* pszSql )`	Prepares SQL statement and returns SQLiteStatement object.	
`void BeginTransaction()` `void CommitTransaction()` `void RollbackTransaction()`	Transaction begining, commiting and rollbacking.	`db.BeginTransaction()` is shortcut to `db.ExecuteNonQuery(_T("BEGIN TRANSACTION"));` etc.
`static _tstring get_VersionString()`	SQLite3 library version string.	`tcout << _T("Hello from SQLite3 version ") << SQLiteConnection::get_VersionString() << endl;`
`static int get_VersionNumber()`	SQLitte3 library version number.	`tcout << _T("Hello from SQLite3 version ") << SQLiteConnection::get_VersionNumber() << endl;`

Operators.

Operator	Description	Sample code / Comments
`operator sqlite3*() const`	Access to `sqlite3*` handle.	`sqlite3* pDb = db;`
`operator bool() const`	Test if database is open.	

SQLiteStatement class.

This is a wrapper class around `sqlite3_statement*` handle and represents prepared SQL statement.

In most cases you do not create this object explicitly but use `SQLiteConnection::Prepare` method to build this object.

Constructors.

Constructor	Description	Sample code / Comments
`SQLiteStatement( SQLiteConnection& connection )`	Initializes empty object.	`SQLiteStatement stmt(db);` `...` `stmt.Prepare(_T("SELECT * FROM Test"));`
`SQLiteStatement( SQLiteConnection& connection, const TCHAR* pszSql )`	Initializes object and prepares `pszSql` statement. If statement preparation fails SQLiteException is thrown.	`SQLiteStatement stmt( db, _T("SELECT * FROM Test") );`
`SQLiteStatement( SQLiteConnection& connection, const _tstring& sSql )`	nitializes object and prepares `sSql` statement. If statement preparation fails SQLiteException is thrown.	
`SQLiteStatement( SQLiteStatement& stmt )`	Copy constructor. `Stmt` object will be detached form `sqlite3_stmt*` handle.	`SQLiteStatement stmt( db.Prepare(_T("SELECT * FROM Test")) );`

Methods.

Method(s)	Description	Sample code / Comments
`SQLiteConnection& get_Connection() const`	Gets SQLiteConnection object reference associated with this object.	`SQliteConnection& stmt_db = stmt.get_Connection();`
`void Prepare( const _tstring& sSql )` `void Prepare( const TCHAR* pszSql )`	Prepares SQL statement. If statement compilation fails SQLiteException is thrown.	`stmt.Prepare(_T("SELECT * FROM Test"));`
`void Reset()`	Resets previously prepared statement to its initial state, ready to re-executed.	`stmt.Reset()`
`void Finalize()`	Deletes previously prepared statement. Releases `sqlite3_stmt*` handle.	Destructor also finalizes prepared statement if you do not call this method.
`int get_ParameterCount() const`	Number of statement parameters. Statement must be prepared.	
`_tstring get_ParameterName( int nParam ) const`	Gets name of `nParam`-th parameter. Use only when you use named parmaters. **Parameters are numbered from 1!**	`int nParamCount = stmt.get_ParameterCount();` `for( int i=1; i<=nParamCount; i++ )` `{` `tcout << _T("Parameter ") << i << _T(": ") << stmt.get_ParameterName(i) << endl;` `}`
`void BindBlob( int nParam, const void* pBlock, int nSize )` `void BindBlob( const _tstring& sParam, const void* pBlock, int nSize )` `void BindBlob( const TCHAR* pszParam, const void* pBlock, int nSize )`	Binds BLOB to statement's parameter. You may specify parameter by index (`nParam`) or by name (`sParam,pszParam`).	`static const BYTE blob[4] = { 0x01, 0x02, 0x03, 0x04 };` `...` `stmt.BindBlob( 1, blob, 4 )` `stmt.BindBlob( _T(":blob_parameter"), blob, 4 );`
`void BindDouble( int nParam, double val )` `void BindDouble( const _tstring& sParam, double val )` `void BindDouble( const TCHAR* pszParam, double val )`	Binds floating-point value to statement's parameter. You may specify parameter by index (`nParam`) or by name (`sParam,pszParam`).	`stmt.BindDouble( 2, 1.7888888 );` `stmt.BindDouble( _T(":double_parameter"), 1.7888888 );`
`void BindInt( int nParam, int val )` `void BindInt( const _tstring& sParam, int val )` `void BindInt( const TCHAR* pszParam, int val )`	Binds integer value to statement's parameter. You may specify parameter by index (`nParam`) or by name (`sParam,pszParam`).	`stmt.BindInt( 3, 1234 );` `stmt.BindInt( _T(":int_parameter"), 1234 );`
`void BindInt64( int nParam, sqlite_int64 val )` `void BindInt64( const _tstring& sParam, sqlite_int64 val )` `void BindInt64( const TCHAR* pszParam, sqlite_int64 val )`	Binds 64-bit integer value to statement's parameter. You may specify parameter by index (`nParam`) or by name (`sParam,pszParam`).	`stmt.BindInt64( 4, 1234123456 );` `stmt.BindInt64( _T(":int64_parameter"), 1234123456 );`
`void BindText( int nParam, const TCHAR* val )` `void BindText( int nParam, const _tstring& val )` `void BindText( const _tstring& sParam, const TCHAR* val )` `void BindText( const _tstring& sParam, const _tstring& val )` `void BindText( const TCHAR* pszParam, const TCHAR* val )` `void BindText( const TCHAR* pszParam, const _tstring& val )`	Binds text to statement's parameter. You may specify parameter by index (`nParam`) or by name (`sParam,pszParam`).	`tostringstream ss;` `ss << _T("-=<") << hex << rand() << _T(">=-");` `stmt.BindText( 5, ss.str() );` `stmt.BindText( _T(":str_parameter"), SQLiteConnection::get_VersionString() )`

Method(s)	Description	Sample code / Comments

`void BindNull( int nParam )`
`void BindNull( const _tstring& sParam )`
`void BindNull( const TCHAR* pszParam )`
	Binds NULL value to statement's parameter.
	You may specify parameter by index (`nParam`) or by name (`sParam`,`pszParam`).
		`stmt.BindNull( 6 );`
		`stmt.BindNull( _T(":nil_parameter") );`

`int get_ColumnCount() const`	Returns the number of columns in the result set returned by the prepared statement.	

`_tstring get_ColumnName( int nColIdx )`	This function returns the column heading for the `nColIdx`-th column of prepared statement.	`int nColumnCount = stmt.get_ColumnCount();`
		`for( int i=0; i<nColumnCount; i++ )`
		`{`
		`  tcout << _T("Column ") << 1 << _T(": ") << stmt.get_ColumnName(i) << endl;`
		`}`

`_tstring get_ColumnDecltype( int nColIdx )`	Returns declared type of `nColIDx`-th column.	`tcout << _T("Declared column type: ") << stmt.get_ColumnDecltype(0) << endl;`

`bool Step()`	One step execution of prepared statement. One step return one row. Returns true if new row is fetched and flase when end of record set was reached.	`SQLiteStatement stmt( db, _T("SELECT * FROM TEST") );`
		`while( stmt.Step() )`
		`{`
		`  // dump data here`
		`}`

`void Execute()`	Non-query statement execution. After execution statement is ready to re-execute.	`SQLiteStatement stmt( db, _T("INSERT INTO Table(a) VALUES(?)") );`
		`for( int i=0; i<100; i++ )`
		`{`
		`stmt.BindInt( 1, i );`
		`stmt.Execute();`
		`}`
		
		Instead of calling `Execute()` method you can use following sequence:
		
		`stmt.Step();`
		`stmt.Reset();`

`int get_DataCount() const`	Returns the number of values in the current row of the result set. You must call Step method before.	

`const void* get_ColumnBlob( int nColIdx, int& nBlobSize ) const`	Returns BLOB and its size of `nColIdx`-th column in the current row of the result set.	`int nBlobSize;`
		`const void* pBlob = stmt.get_ColumnBlob(0,nBlobSize);`
		`tcout << _T("Blob size: ") << nBlobSize << endl;`

`double get_ColumnDouble( int nColIdx ) const`	Returns floating-point value of `nColIdx`-th column in the current row of the result set.	
`int get_ColumnInt( int nColIdx ) const`	Returns integer value of `nColIdx`-th column in the current row of the result set.	
`sqlite_int64 get_ColumnInt64( int nColIdx ) const`	Returns 64-bit integer value of `nColIdx`-th column in the current row of the result set.	
`_tstring get_ColumnText( int nColIdx ) const`	Returns text value of `nColIdx`-th column in the current row of the result set.	

Method(s)	Description	Sample code / Comments

`int get_ColumnType( int nColIdx ) const`	Returns type of `nColIdx`-th column in the current row of the result set.
	Possible values are (values taken from `sqlite3.h` header):
	
	`SQLITE_INTEGER = 1`
	`SQLITE_FLOAT = 2`
	`SQLITE_TEXT = 3`
	`SQLITE_BLOB = 4`
	`SQLITE_NULL = 5`
		`switch( stmt.get_ColumnType(n) )`
		`{`
		`case SQLITE_INTEGER:`
		`tcout << _T("INTEGER: ") << stmt.get_ColumnInt(n) << endl;`
		`break;`
		`case SQLITE_FLOAT:`
		`tcout << _T("FLOAT: ") << stmt.get_ColumnDouble(n) << endl;`
		`break;`
		`case SQLITE_TEXT:`
		`tcout << _T("TEXT: ") << stmt.get_ColumnText(n) << endl;`
		`break;`
		`case SQLITE_BLOB:`
		`tcout << _T("BLOB") << endl;`
		`break;`
		`case SQLITE_NULL:`
		`tcout << _T("NULL") << endl;`
		`break;`
		`}`

`bool IsNull( int nColIdx ) const`	Tests if `nColIdx`-th column in the current row of the result set has NULL value.	`if ( stmt.IsNull(n) ) tcout << _T("NULL value") << endl;`

Operators.

Operator	Description	Sample code / Comments
`operator sqlite3_stmt*() const`	Access to `sqlite3_stmt*` handle.	`sqlite3_stmt* pStmt = stmt;`
`operator bool() const`	Test if object is prepared.	

Library usage.

Most of library stuff is included in `LiteX.hpp` header file. Few functions are implemented in `LiteX.cpp` file. To use this library in your project simply add these two files to your project and use:

```
#include "LiteX.hpp"
```

directive in every module you want to use this library. There's no LIB nor DLL file. That's because most of class methods are inline. All classes and functions are groupped into `litex` namespace. Your project also must have access to `sqlite3.h` header from SQLite3 package.

All files you can find in LiteX package in `LiteX_pp` subdirectory. In this directory you can also find simple console application that demonstrates how to use LiteX++ library.

This library works only on Windows platform. Porting to other platforms is possible and requires text encoding routines change only.This library was tested with Visual Stusio C++ 6.0 (project files included) and Visual Studio C++ .NET 2003/2005 compiler (only 2005 project files included).

Using LiteX++ together with LiteX Automation.

When you register LiteX Automation library information about location of `sqlite3.dll` library is stored into registry. LiteX Automation library exports all native SQLite3 functions.

If you create application that uses dynamically linked native SQLite3 API you must put another copy of `sqlite3.dll` into the directory where this application resides (best solution) or into directory specified by `PATH` enviroment variable. This second version of DLL is unnecessary but your application must read LiteX Automation library location from registry. You may by hand call **LoadLibray** and then **GetProcAddress** functions but writing C++ applications you rather use header file with function definitions (`sqlite3.h`) and static import library (`sqlite3.lib`).

If you specify sqlite3.dll in your C++ project as delay-loaded library (`/DELAYLIB` linker option) you have control how to load delay-loaded DLLs by own notification hooks. LiteX++ library currently implements such simple notification hook that reads LiteX Automation library location and load this DLL if nessesary. This even works with Unicode version of LiteX Automation (sqlite3u.dll). There are 3 functions into `litex::delayload` namespace:

Function	Description	Sample code
`void set_handler();`	Installs own delay-load DLL notification hook. When `sqlite3.dll` library is needed it tries to locate LiteX Automation library. If LiteX Automation library cannot be found standard search procedure will be used.	`using namespace litex;` `delayload::set_handler();`
`HRESULT load_library( bool bAutomation = true );`	Loads `sqlite3.dll` library. If `bAutomation` is **true** `set_handler()` is called before. You may call this routine before any call to SQLite3 engine to make sure that this engine is accessible. 0 means success, `0x8007007e` is the most common error code and means that `sqlite3.dll` library cannot be found. **Warning:** Don't call this routine from **DllMain** function!	`using namespace litex;` `delayload::load_library(true);`
`bool free_library();`	Unloads previously loaded `sqlite3(u).dll` library by `load_library()` function. You don't need to call this function.	`using namespace litex;` `delayload::free_library();`

These functions are defined only if **`_LITEX_WITH_DELAYLOAD`** macro is defined. Note that using this functions makes only sense if you set `sqlite3.dll` library as **delay-loaded DLL**. Do not use these functions if SQLite3 API is linked statically in your project. By using these functions you can still use header and import library without calling **LoadLibrary** and **GetProcAddress** by hand. All you need is to call `set_handler()` or `load_library()` function at the beginning of your application (library) before any call to SQLite3 API. Look into `LiteX_pp` subdirectory to see how this technique works.

LiteX ADO .NET.

This ADO .NET provider is my try to create ADO .NET provider for SQLite3 databases.
This projest is in very early development state. Currently there's no documentation for this provider so you must look into sources.

This provider is creating using *Visual Studio .NET 2005*. This is **C++** project (C++ with managed extensions, not C#) using new syntax for managed extensions. Because new syntax is used it cannot be compiled using *Visual Studio .NET 2003* or any earlier version of Visual Studio. This is not pure managed code - it uses SQLite3 native C API. This API may be linked statically or dynamically (from DLL).

Sources and binaries of this provider you can find in `LiteX_NET` subdirectory of LiteX package. `litextest` and `litexgtest` subdirectories of this package contains sample C# console and GUI applications that demonstrates basic features of this provider.

Currently sources of LiteX ADO .NET are removed from source packages. They are available only from SVN.

Last modification time: 2008-04-16 10:24.