NAME

phasor_sqlite − SQLite database interface for Phasor

SYNOPSIS

sqlite_open(path)
sqlite_close(db_handle)
sqlite_exec(db_handle, sql)
sqlite_prepare(db_handle, sql)
sqlite_step(stmt_handle)
sqlite_column(stmt_handle, column_index)
sqlite_finalize(stmt_handle)
sqlite_free_string(string_handle)

DESCRIPTION

The Phasor SQLite plugin provides native bindings to the SQLite database engine, enabling Phasor scripts to create, query, and manage SQLite databases. The plugin uses a handle-based system for managing database connections and prepared statements.

Database and statement handles are integers that reference internal resources managed by the plugin. Applications must properly close databases and finalize statements to prevent resource leaks.

DATABASE FUNCTIONS

sqlite_open(path)

Open or create an SQLite database file.

Arguments:

path	Path to the database file. If the file does not exist, SQLite will create it

Returns: Integer database handle on success, or null on failure

sqlite_close(db_handle)

Close an open database connection.

Arguments:

db_handle	Database handle returned by **sqlite_open()**

Returns: Boolean true if database was closed successfully, false if handle was invalid

Notes: Automatically releases the database handle from the internal handle table

sqlite_exec(db_handle,**sql)**

Execute a SQL statement that does not return data (e.g., CREATE, INSERT, UPDATE, DELETE).

Arguments:

db_handle	Database handle
sql	SQL statement to execute

Returns: Boolean true on success, false on failure

Notes: This function is suitable for DDL and DML statements that do not return result sets. For queries that return data, use **sqlite_prepare()** and **sqlite_step()** instead

PREPARED STATEMENT FUNCTIONS

sqlite_prepare(db_handle,**sql)**

Prepare a SQL statement for execution.

Arguments:

db_handle	Database handle
sql	SQL statement to prepare

Returns: Integer statement handle on success, or null on failure

Notes: Prepared statements must be finalized with **sqlite_finalize()** after use to prevent resource leaks

sqlite_step(stmt_handle)

Execute one step of a prepared statement.

Arguments:

stmt_handle Statement handle returned by **sqlite_prepare()**

Returns: Boolean true if a row is available (SQLITE_ROW), false if execution is complete (SQLITE_DONE), or null on error

Notes: Call repeatedly to iterate through all result rows. When true is returned, use **sqlite_column()** to retrieve column values from the current row

sqlite_column(stmt_handle,**column_index)**

Retrieve a column value from the current row.

Arguments:

stmt_handle Statement handle

column_index
Zero-based column index

Returns: Column value with appropriate type (integer, float, string, or null), or null if column index is out of range or statement handle is invalid

Notes: For string values, the plugin stores the string internally and returns a pointer to it. The string remains valid until the next call to **sqlite_step()** or **sqlite_finalize()**

sqlite_finalize(stmt_handle)

Finalize a prepared statement and release its resources.

Arguments:

stmt_handle Statement handle to finalize

Returns: Boolean true if statement was finalized successfully, false if handle was invalid

Notes: Always call this function after finishing with a prepared statement. Automatically releases the statement handle from the internal handle table

UTILITY FUNCTIONS

sqlite_free_string(string_handle)

Free a string stored in the internal string table.

Arguments:

string_handle
String handle to free

Returns: Null value

Notes: This function is primarily for internal memory management and typically does not need to be called by user code

EXAMPLES

Opening and Closing a Database

```
// Open database
var db = sqlite_open("mydata.db");
if (db == null) {
    puts("Failed to open database");
    return;
}

// Use database...
// Close database
sqlite_close(db);
```

Creating a Table

```
var db = sqlite_open("users.db");

var sql = "CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER
)";

if (sqlite_exec(db, sql)) {
    puts("Table created successfully");
} else {
    puts("Failed to create table");
}

sqlite_close(db);
```

Inserting Data

```
var db = sqlite_open("users.db");

var sql = "INSERT INTO users (name, age) VALUES ('Alice', 30)";
sqlite_exec(db, sql);

sql = "INSERT INTO users (name, age) VALUES ('Bob', 25)";
sqlite_exec(db, sql);

sqlite_close(db);
```

Querying Data

```
var db = sqlite_open("users.db");

var stmt = sqlite_prepare(db, "SELECT id, name, age FROM users");
if (stmt == null) {
    puts("Failed to prepare statement");
    sqlite_close(db);
    return;
}

// Iterate through results
while (sqlite_step(stmt)) {
    var id = sqlite_column(stmt, 0);
    var name = sqlite_column(stmt, 1);
    var age = sqlite_column(stmt, 2);

    putf("ID: %d, Name: %s, Age: %d", id, name, age);
}

sqlite_finalize(stmt);
sqlite_close(db);
```

Complete CRUD Example

```
var db = sqlite_open("inventory.db");
```

```
// Create
sqlite_exec(db, "CREATE TABLE items (id INTEGER PRIMARY KEY,
        name TEXT, quantity INTEGER)");

// Insert
sqlite_exec(db, "INSERT INTO items (name, quantity)
        VALUES ('Widget', 100)");
sqlite_exec(db, "INSERT INTO items (name, quantity)
        VALUES ('Gadget', 50)");

// Read
var stmt = sqlite_prepare(db, "SELECT * FROM items WHERE quantity > 25");
while (sqlite_step(stmt)) {
  putf("%s: %d", sqlite_column(stmt, 1), sqlite_column(stmt, 2));
}
sqlite_finalize(stmt);

// Update
sqlite_exec(db, "UPDATE items SET quantity = 75 WHERE name = 'Gadget'");

// Delete
sqlite_exec(db, "DELETE FROM items WHERE quantity < 60");

sqlite_close(db);
```

NOTES

- All database and statement handles are integers managed by internal hash tables
- Column indices in **sqlite_column()** are zero-based
- The plugin automatically handles type conversion between SQLite types and Phasor types
- Strings returned by **sqlite_column()** are stored in an internal string table and remain valid until the statement is finalized
- Always finalize prepared statements and close databases to prevent resource leaks
- The plugin is thread-safe, using mutexes to protect internal data structures
- BLOB data types are not currently supported
- Parameter binding is not currently supported; use string concatenation to build dynamic queries (be aware of SQL injection risks)

ERRORS

Functions return null or false on error. Common error conditions include:

- Invalid file path or permissions when opening database
- SQL syntax errors in **sqlite_exec()** or **sqlite_prepare()**
- Invalid database or statement handles
- Out-of-range column indices in **sqlite_column()**

SEE ALSO

phasor-ffi(7), **sqlite3**(1)

SQLite documentation: https://www.sqlite.org/docs.html

AUTHOR

Daniel McGuire

COPYRIGHT

Copyright © 2026 Daniel McGuire