SQL Databases in PicoMite MMBasic

MMBasic can keep data in an SQL database: a file on the SD card or a USB drive holding tables of rows, which a program adds to, changes, searches and sorts with statements in SQL, the standard language for databases. The database engine is SQLite, the same engine found in phones, browsers and aircraft, and a database file made on the PicoMite can be read by SQLite on a PC, or on a Colour Maximite 2, and the other way round. A database made on a PC opens whether it keeps its text as UTF-8 (the usual) or as UTF-16; text reaches BASIC as UTF-8.

The commands are the same as on the Colour Maximite 2, so a program that uses them runs on either.

Which PicoMites

SQL is in the builds that run loadable programs: the PicoMitePC3 and the PicoMiteHDMIUSB, with PSRAM enabled.

SQLite is not part of the firmware. It is a module, the file sql.mod, which MMBasic loads when a program opens a database and unloads when it closes it. It is in the sql folder of the PicoMite programs zip that comes with the firmware: copy that folder to the root of the A: drive (A:/sql/sql.mod) or of the SD card (B:/sql/sql.mod). SQL OPEN looks in A:/sql first, then in B:/sql.

A database must be on B: (the SD card) or C: (a USB drive). The A: drive is a littlefs file system in the flash, which rewrites a file from the block that changed to its end, so each commit would rewrite most of a database; SQL OPEN refuses a database there ("SQL: a database must be on B: or C:"). The module itself may be on A:.

The commands for it are SQL and the function SQL(, described below and in the Commands and Functions sections of the User Manual. This guide shows how to use them; the SQL language itself is described in many books and on the SQLite web site (sqlite.org/lang.html).

Memory

While a database is open the engine takes about 2.6MB of the heap, in the PSRAM: the module (about 530KB) and the memory SQLite works in (its pool, 2MB unless SQL OPEN gives another size). It also has a stack of its own, 16KB, in the SRAM heap if there is a block of that size free, else in the PSRAM. SQL CLOSE gives it all back.

Opening and Closing

```
SQL OPEN "B:/data/club.db"
...
SQL CLOSE
```

SQL OPEN opens the database file, and makes it (empty) if it is not there. Only one database is open at a time: opening another closes the one before. A file name without a drive or folder is in the current directory, as with other files (which must then be on B: or C:). SQL CLOSE closes it, and gives back the memory the engine used.

The database stays open until SQL CLOSE, or until the program is run again (RUN), cleared (NEW) or another one loaded (or CHAIN): each of these closes it, and any transaction not finished is undone. A program that ends without SQL CLOSE leaves it open at the command prompt.

SQL OPEN can also be given the engine's pool, in kilobytes (256 to 16384, default 2048), and how much of it to use as the cache of the database's pages: SQL OPEN f$, poolKB, cacheKB. Neither is normally needed; a smaller pool leaves more of the heap to BASIC.

Running Statements

SQL Databases in PicoMite MMBasic

SQL EXEC runs SQL statements: one, or several separated by semicolons.

```
SQL EXEC "CREATE TABLE IF NOT EXISTS member(id INTEGER PRIMARY KEY, name TEXT, born INTEGER)"
SQL EXEC "INSERT INTO member(name, born) VALUES('Ann', 1961)"
```

Values from the program go in as parameters: each ? in the statement is replaced by the next value after it, a number or a string. There is then no need to put a string's quotes into the statement, and no harm in a value that has quotes in it.

```
SQL EXEC "INSERT INTO member(name, born) VALUES(?, ?)", n$, year%
SQL EXEC "UPDATE member SET born = ? WHERE name = ?", 1962, "Ann"
```

SQL(ROWS) is the number of rows the last INSERT, UPDATE or DELETE changed, and SQL(ID) the key of the last row inserted.

Each statement on its own is a transaction: its changes are on the card, safe, when it ends. That takes about 20ms. Many changes together are much faster inside one transaction, which is written to the card once at its end:

```
SQL EXEC "BEGIN"
FOR i = 1 TO 1000
  SQL EXEC "INSERT INTO reading(t, value) VALUES(?, ?)", i, v(i)
NEXT i
SQL EXEC "COMMIT"
```

A thousand inserts take a fraction of a second like this, against 20 seconds one by one. ROLLBACK in place of COMMIT undoes them all. If the power fails during a transaction, the next SQL OPEN finds it unfinished and undoes it, so the database is never left half changed.

Queries

SQL QUERY starts a query, and SQL(NEXT) moves to its next row: 1 if there is one, 0 when there are no more. SQL(n) is the value of column n of the row (the first column is 1), as an integer, a floating point number or a string, as the column holds it.

```
SQL QUERY "SELECT name, born FROM member WHERE born > ? ORDER BY born", 1950
DO WHILE SQL(NEXT)
  PRINT SQL(1), SQL(2)
LOOP
```

SQL(COLUMNS) is the number of columns, SQL(NAME n) the name of column n. A column with no value (SQL's NULL) gives 0, or an empty string for a text column; SQL(ISNULL n) is 1 if column n has no value. SQL(TYPE n) says what SQLite holds in column n of the row: 1 an integer, 2 a floating point number, 3 text, 4 a BLOB, 5 no value (sqlite3_column_type()'s numbers). It tells a BLOB from text and 2.0 from 2, which SQL(n) gives alike, so a program that writes the data out again, as a dump does, can keep each value's type.

A query holds memory, and its tables, until it has given all its rows. A query read only in part (one row of a count, say) is ended by SQL CLOSE QUERY, by the next SQL QUERY, or by SQL CLOSE. A statement that needs a table such a query still holds (DROP or CREATE INDEX on it) ends the query first.

SQL(SELECT puts all the rows of a query into an array at once, and returns how many it put there. A one dimensional array takes the first column; a two dimensional array (row, column) takes as many columns as it has. A string array takes each value as text. It stops when the array is full.

```
DIM names$(100)
n = SQL(SELECT "SELECT name FROM member ORDER BY name", names$())
DIM t$(20, 2)
n = SQL(SELECT "SELECT name, born FROM member WHERE born < ?", t$(), 1970)
```

SQL Databases in PicoMite MMBasic

Long Text

An MMBasic string holds at most 255 characters, so SQL(n) gives an error for a longer text value (SQL: a string of 300 characters, longer than 255); the query stays on its row, so a program that catches the error can read the column again as a LONGSTRING. A LONGSTRING, an integer array that holds text of any length (see LONGSTRING in the Commands), takes it instead:

```
DIM note%(1000)                     ' a LONGSTRING of up to 8,000 characters
LONGSTRING CLEAR note%()
LONGSTRING APPEND note%(), "The first part of a long note, "
LONGSTRING APPEND note%(), "and the rest of it."
SQL EXEC "INSERT INTO note(body) VALUES(?)", note%()
SQL QUERY "SELECT body FROM note"
DO WHILE SQL(NEXT)
  n = SQL(LONGSTRING 1, note%())    ' the text into note%(), its length in n
  PRINT n, LGETSTR$(note%(), 1, 60)
LOOP
```

- SQL(LONGSTRING n, a%()) puts column n of the row into the LONGSTRING a%(), and returns its length. A%() too small for it is an error.
- A LONGSTRING among a statement's values gives its text to a ? parameter.
- A statement can itself be a LONGSTRING, for one longer than 255 characters: SQL EXEC s%(), SQL QUERY s%(), SQL(SELECT s%(), ...).

Strings in arrays and structures still hold only as many characters as their length.

Structures and Tables

A table's row and an MMBasic structure are much the same thing, and SQL can move rows into and out of structures by matching the names of the structure's members with the names of the table's columns.

```
TYPE Member
  id AS INTEGER
  name AS STRING LENGTH 40
  born AS INTEGER
END TYPE
DIM m AS Member, list(100) AS Member

SQL CREATE Member                   ' a table MEMBER with the type's members as its columns
m.name = "Ann" : m.born = 1961
SQL INSERT m                        ' a row from m; m.id is now its key
SQL INSERT list()                   ' a row for each element of the array
SQL QUERY "SELECT * FROM member ORDER BY name"
DO WHILE SQL(FETCH m)               ' as SQL(NEXT), the row put into m
  PRINT m.name, m.born
LOOP
n = SQL(SELECT "SELECT * FROM member WHERE born > ?", list(), 1950)
SQL EXEC "UPDATE member SET born = :born WHERE id = :id", m
```

- SQL CREATE type makes the table if it is not there, named after the type (or SQL CREATE type, table$). INTEGER members are INTEGER columns, FLOAT members REAL and STRING members TEXT; a member named ID that is an integer is the table's key.
- SQL INSERT var adds a row from a structure, or a row for each element of an array of structures (var()), in one

transaction: if it stops part way, through an error or Ctrl-C, none of its rows are kept. When the ID member is 0, the database gives the row a new key and puts it into the member.

- SQL(FETCH var) and SQL(SELECT with an array of structures fill each member from the column of its name (in any case); members with no column of their name are cleared.
- A statement given a structure among its values takes :name parameters from the members of that name.
- A nested structure's members are columns named PARENT_CHILD, and an array member's elements columns NAME_0, NAME_1 and so on (NAME_1 first under OPTION BASE 1).

An Example

The sql folder of the programs zip has a sample database, computers.db, with a table of home computers, and this program, computers.bas, which uses it. The table's columns have the names of the members of the type Computer, so each row goes straight into a structure. (A member's name cannot be a command's, such as CPU: a line inside TYPE that starts with one is read as the command.)

```
TYPE Computer
  id AS INTEGER
  name AS STRING LENGTH 30
  maker AS STRING LENGTH 20
  year AS INTEGER
  processor AS STRING LENGTH 10
  ramkb AS INTEGER
END TYPE
DIM c AS Computer, z80(20) AS Computer
DIM INTEGER i, n

SQL OPEN "B:/sql/computers.db"

PRINT "By year:"
SQL QUERY "SELECT * FROM computer ORDER BY year, name"
DO WHILE SQL(FETCH c)
  PRINT c.year, c.name, c.maker
LOOP

n = SQL(SELECT "SELECT * FROM computer WHERE processor = ? ORDER BY ramkb DESC", z80(), "Z80")
PRINT : PRINT n; " with a Z80:"
FOR i = 0 TO n - 1
  PRINT z80(i).name, z80(i).ramkb; "K"
NEXT i

c.id = 0 : c.name = "ZX80" : c.maker = "Sinclair" : c.year = 1980 : c.processor = "Z80" : c.ramkb
= 1
SQL EXEC "DELETE FROM computer WHERE name = ?", c.name
SQL INSERT c
PRINT : PRINT "The ZX80 added as number"; c.id

PRINT : PRINT "Makers:"
SQL QUERY "SELECT maker, count(*) FROM computer GROUP BY maker ORDER BY 2 DESC, 1"
DO WHILE SQL(NEXT)
  PRINT SQL(1), SQL(2)
LOOP
SQL CLOSE
```

It lists the computers in the order they came out, finds those with a Z80 processor (filling an array of the type), adds a row from a structure (SQL INSERT gives it its key in c.id; the DELETE before it stops a second run adding it twice) and counts the computers of each maker.

Errors

A statement the database cannot run is an MMBasic error, "SQL: " followed by SQLite's own message (SQL: no such table: member), which a program can catch with ON ERROR SKIP and MM.ERRMSG$.

Ctrl-C stops a long statement with the error "SQL: stopped by Ctrl-C". It is an error like any other: ON ERROR SKIP catches it and the program goes on, so a program can let its user stop a long query and carry on. Without ON ERROR the program stops with it, as at any Ctrl-C. A statement stopped while it was changing the database leaves no change behind, and if it was in a transaction (after BEGIN) SQLite undoes the whole transaction. Ctrl-C between the statements of one SQL EXEC stops it in the same way. The key is the one OPTION BREAK sets (Ctrl-C unless it is changed); with OPTION BREAK 0 nothing stops a statement.

Speed

The SD card is the limit for a large database: each page fetched from a random place costs a read from the card. The engine keeps the pages it has used in memory (its cache, 256KB with the default pool), so a database that fits in the cache works at the speed of memory once it has been read. Measured on a PicoComputer 3 with the database on its SD card:

Operation	PicoComputer 3
Opening a database	27ms
One insert committed on its own	18ms
1,000 inserts in one transaction	0.14s
1,000 rows found by an index	0.17s
50,000 rows inserted in one statement	17s
Sorting those 50,000 rows (ORDER BY)	5.8s
CREATE INDEX on them	7.4s
PRAGMA integrity_check of them	12.5s

Indexes make the difference: put one on the columns a program searches, sorts or joins by.

Limits

- One database open at a time (ATTACH reaches more through it), on B: or C:.
- A statement in a string has at most 255 characters (a longer one can be a LONGSTRING); up to 20 values after it.
- A structure used with SQL can have at most 64 columns, counting nested members and array elements; a string from the database longer than the variable or member it goes into is an error (SQL(LONGSTRING) reads text of any length).
- During a large sort the engine writes temporary files to the root of the database's drive (sqlite_....tmp), and removes them after.