RDBMS4C

Relational Database Management System for C language

reference guide

__UNDER CONSTRUCTION

Author: Szabolcs Kiss

Introduction

The **RDBMS4C** solution is an embedded, in memory database, designed for C developers. The original motivation was to improve the possibilities of the C language, keeping all of the existing low level advantages.

What is RDBMS4C

Embedded, in memory database

C API

NOSQL

Open source (GPL)

Dynamic tables

C data types for columns

Foreign key support

Primary, unique and normal indexes

Column and function based indexes

Cursors for DML and query

Data safety

String support C developers

What RDBMS4C is not

This is not general database, this is a C language extension via API

This is not general database, there is no persistent data on disc

Currently this is not a tested product, this is a research and development style project

RDBMS4C core

Build up your database structure (DDL)

Standard data types

The data types reflect the well known C data types:

Data type	Related C data type
__char	char
__unsigned_char	unsigned char
__int	int
__unsigned_int	unsigned int
__long	long
__unsigned_long	unsigned long
__float	float
__double	double

Custom data types

Data type	Description
__void_pointer	General pointer
__char_array	String, char* in C
__foreign_key	Foreign key

Record definitions

To describe the fileds for table creation, you need to have a variable with the following type:

```
__db_record_definition rec_def;
```

First of all, you need to create the record definition variable with the number of the fields:

```
rec_def = db_create_record_definition(num_of_fields);
```

function parameters:

0 - **[int]** num of fields

After this step, you can set up the fields:

```
db_set_record_definition_field(
    rec_def, field_index, "id", __NOT_NULL, __unsigned_int);
```

function parameters:

0 – **[__db_record_definition]** record definition variable

1 – **[int]** zero based index of the field you set up

2 – **[char*]** name of the field

3 – **[__db_field_NULL_definition]** mandatory of the field, **__NOT_NULL** or **__NULL**

4 – **[__db_field_data_type]** data type, possible values are described above

Primary keys

The most important thing concerning primary keys is, that you can't create table without primary key. This forces the keeping of the RDBMS normal format. So prior to table creation you must create a primary key for the table, which is basically a custom index. This index is placed at the 0th index position and you can't delete or override this. First of all, you need a primary key variable:

```
__db_primary_key primary_key;
```

The primary key creation is the following (in case of complex primary key the parameter can be > 0):

```
primary_key = db_create_pk(num_of_pk_fields);
```

function parameters:

0 - **[int]** num of fields in the index

You also need to set up the fields for the primary key (in case of complex primary key, you need to call this for all of the key fields):

```
db_set_index_field(primary_key, index_field, field_index, __asc);
```

function parameters:

0 – **[__db_primary_key]** the primary key variable

1 – **[int]** the position of the index field you set up

2 – **[int]** the position of the field in the record definition

3 – **[__db_index_field_sort_type]** sort type, **__asc** or **__desc**

Tables

You need the following variable to create a table:

```
__db_table table;
```

Having the record definition and primary key, you can create the table with the following calling:

```
table = db_create_table(rec_def, primary_key);
```

function parameters:

0 – **[__db_record_definition]** the record definition

1 – **[__db_primary_key]** the primary key variable

If you don't need the table anymore, you need to drop that with the following statement:

```
db_drop_table(table);
```

function parameters:

0 – **[__db_table]** the table to be dropped

! Note, that all of the included components (indexes, record definitions, etc) will be destroyed.

! Note, that child records and child tables are not maintained in this statement.

Auto incremented primary key

This is a so important functionality regarding comfort and performance. If you use an unsigned long type as primary key, you can set up your table as auto incremented. It results, that you don't need to ensure the unique values at the insert statements (you can pass a simple 0 value as it will be overwritten), but even more important, that there won't be unique checking performed prior to insert the record, as the table ensures the unique value internally. This results the good performance (if there is no any other unique index on the table).

You can simple call the following statement once the table is done (and you ensured the unsigned long primary key).

db_table_auto_increment(table);

function parameters:

0 – **[__db_table]** table to be auto incremented

Indexes

You can create different indexes on your table, but important to remember, that you already created one as primary key prior to create the table.

The possible index types are the followings:

- normal index
- unique index
- primary index

Basically the indexes are based on fileds from the table, however you also can create index which is based on your custom function.

The next steps show the index creation process, first of all, you need a variable:

```
__db_index index;
```

```
index = db_create_index(num_of_index_fields, __normal);
```

function parameters:

0 – **[int]** num of the fields in the index

1 – **[__db_index_index_type]** type of the index: **__normal** or **__unique** (the **__primary** is not usable for general indexes)

```
db_set_index_field(index, index_field, field_index, __asc);
```

function parameters:

0 – **[__db_index]** the index variable

1 – **[int]** the position of the index field you set up

2 – **[int]** the position of the field in the record definition

3 – **[__db_index_field_sort_type]** sort type, **__asc** or **__desc**

```
db_set_table_index(table, field_index, index);
```

function parameters:

0 – **[__db_table]** the table variable

1 – **[int]** the position of the index in the table (starting from 1, as 0 is the primary index)

2 – **[__db_index]** the index variable

Function based indexes

You also can create indexes based on user function instead of table columns.

The index function function needs to be implemented with the following interface:

```
typedef int __db_index_function_based(
   const __db_record_s_record* p_record_1,
   const __db_record_s_record* p_record_2);
```

You will receive the records in this function to be compared when the index works. You should return with the result of the comparison (-1, 0, 1).

```
__db_index index;

index =
db_create_function_based_index(__normal, function);
```

function parameters:

0 – **[__db_index_index_type]** type of the index: **__normal** or **__unique** (the **__primary** is not usable for general indexes)

1 - **[__db_index_p_function_based]** user defined function based index

Relationship between tables (foreign keys)

The well known foreign key solution has been implemented as data type in the record definition, so you can build up the connection between the tables in three steps:

1. When you are setting up the record definition of the child table:

```
db_set_record_definition_field(rec_def, field_index,
    "foreign_key_name", __NOT_NULL, __foreign_key);
```

function parameters:

0 – **[__db_record_definition]** record definition variable

1 – **[int]** zero based index of the field you set up

2 – **[char*]** name of the field, in this case the name ofthe foreign key as well

3 – **[__db_field_NULL_definition]** mandatory of the field, in this case the mandatory of the relationship, **__NOT_NULL** or **__NULL**

4 – **[__db_field_data_type]** data type, in case of foreign key it must be the **__foreign_key** constant

2. When both of the child table and parent table are available (child table must have a foreign key field and the parent table must be created with primary key):

```
db_set_foreign_key_table(child_table, field_index, parent_table);
```

function parameters:

0 – **[__db_table]** child table

1 – **[int]** zero based position of the foreign key field in the child table

2 – **[__db_table]** parent table

3. When you are inserting a child record (learn more about insert statement!)**:**

```
db_insert_set_field(child_record_definition, child_fields,
    field_index, parent_record);
```

function parameters:

0 – **[__db_record_definition]** child record definition

1 – **[__db_fields]** child fields

2 – **[int]** zero based position of the foreign key field in the child record definition

3 – **[__db_record]** you need to have the parent record here

Maintaine data (DML)

Insert

The insert process contains three steps. First you need a __db_fields variable prepared by db_insert_preparation function. After that you can set up the fields of the __db_field using the record definition to validate the data. The final step, which inserts the record into the table, needs a valid __db_cursor, which points to the target table.

Step 1:

```
__db_field fields;
fields = db_insert_preparation(num_of_fields);
```

function parameters:

0 – **[int]** number of fields

Step 2:

```
db_insert_set_field(record_definition, fields, field_index,
    value);
```

function parameters:

0 – **[__db_record_definition]** the record definition to check the data

1 – **[__db_fields]** the fields contain the data for insert

2 – **[int]** the position in the field in the fields parameter which will be set up

3 – **[void*]** the pointer of the data which will be used

Step 3:

```
db_insert_into(cursor, fields);
```

function parameters:

0 – **[__db_cursor]** valid cursor pointing to the target table

1 – **[__db_fields]** the fields contain the data for insert

Update

The update process works on field level and like all of the DML processes, update also needs cursor. You can update a field in one step. The statement will affect the record pointed by the cursor.

It means, that the update function assumes, that you have positioned the cursor to the appropriate record.

db_update(cursor, field_index, value);

function parameters:

0 – **[__db_cursor]** valid cursor pointing to the target table and record

1 – **[int]** position of the filed to be updated

2 – **[void*]** the pointer of the data which will be used

If you want to modify a foreign key based field, you can use the following statement:

db_update_foreign_key(cursor, field_index, parent_cursor);

function parameters:

0 – **[__db_cursor]** valid cursor pointing to the target table and record

1 – **[int]** position of the filed to be updated

2 – **[__db_cursor]** cursor, which points to the parent table, the pointed record there will be the new value

Delete

Like every DML statement, the delete function also needs a cursor pointing to the record to be deleted. It means, that the delete function assumes, that you have positioned the cursor to the appropriate record.

db_cursor_delete(cursor);

function parameters:

0 – **[__db_cursor]** valid cursor pointing to the target table and record

! Note, that you can't call this statement inside in a cursor loop.

! Note, that all of the records in the child tables will be deleted as well, so this is a cascade delete.

Query data

Indexes and DML

The indexes are allocated memory areas containing pointer arrays with custom order. The pointers point directly to the records in the memory. It means, that after insert or delete action the indexes needs maintenance or needs to be re-created. To optimalize the performance in these critical cases, the system will destroy the indexes after insert or delete action on the given table. The index will be re-created only after the first query request.

! Note, that after insert or delete action the first query request will be slower as the indexes will be re-created. It means, you should avoid the repeating of DML/QUERY requests, especially in loops.

Cursors

Cursors are dedicated to provide interface for I/O requests, using RDBMS terminology, DML and QUERY requests. A cursor is assigned to a table. One table can have more cursos in the same time and you also can execute parallel loops on cursors pointing to the same table.

A cursor uses a table and one of the table indexes, this index will be used in the cursor loops.

cursor = **db_create_cursor**(table, index_position);

function parameters:

0 – **[__db_table]** the table variable

1 - **[int]** the position of the index in the table

After you decided to drop the cursor, you can you the following statement:

db_drop_cursor(cursor);

function parameters:

0 – **[__db_cursor]** the cursor to be droped

Cursor navigation

You can use the **first**/**last**, **next**/**prev** navigation steps at the cursor. The currently positioned record is available via the **current** function call. All of these navigation functions return with a record variable, if there is a record available, else it returns with null.

```
__db_record record;

record = db_cursor_first(cursor);

record = db_cursor_last(cursor);

record = db_cursor_prev(cursor);

record = db_cursor_next(cursor);

record = db_cursor_current(cursor);
```

function parameters:

0 – **[__db_cursor]** the cursor variable to be targeted

Cursor loop

You can make a FOR-ALL style cursor loop quicky using the built in loop macro statemetns:

__**for_cursor_loop**(record, cursor)

… your statements

__**end_loop**(record, cursor)

function parameters:

0 – **[__db_record]** the record variable containing the currently positioned record

1 – **[__db_cursor]** the cursor variable for the loop

! Note, that you can't execute INSERT or DELETE statement inside in a cursor loop

Access data via cursor record

Once you have found a record via cursor, you can access the fields of the record via the appropriate GET FIELD statements.

For standard data types:

db_get_field_as_char(record, field_position)

db_get_field_as_unsigned_char(record, field_position)

db_get_field_as_int(record, field_position)

db_get_field_as_unsigned_int(record, field_position)

db_get_field_as_long(record, field_position)

db_get_field_as_unsigned_long(record, field_position)

db_get_field_as_float(record, field_position)

db_get_field_as_double(record, field_position)

function parameters:

0 – **[__db_record]** the record variable containing the currently positioned record

1 – **[int]** the position of the field

For custom data types you can use the same parameter structure, but you will get custom results:

db_get_field_as_void(record, field_position)

Returns with a general pointer: void*

db_get_field_as_char_array(record, field_position)

Returns with a general char pointer: char*

db_get_field_as_foreign_key_record(record, field_position)

Returns with a **__db_record** variable containing the parent record (as the foreign keys contain direct pointer to the parent record, this statement can access that directly)

Find a record by key

Based on the fact, that all of the tables you made contains primary key, you can find unique records via keys. You need to create and set up the key prior to use and after that you need to drop the key:

__db_key key;

key = **db_create_key**(cursor);

function parameters:

0 – **[__db_cursor]** the cursor variable

db_set_key_field(key, field_position, value);

function parameters:

0 – **[__db_key]** the key variable to be set up

1 - **[int]** the position of the field

2 - **[void*]** the pointer of the data you want to use

record = **db_find_by_key**(key);

function parameters:

0 – **[__db_key]** the prepared key variable you want to use

db_drop_key(key);

function parameters:

0 – **[__db_key]** the key to be dropped

RDBMS4C extensions

Timer

Timer component provides simple possibility to run a cycle which calls the pre-defined timer entities.

The timer function type definition

The timer function type definition provides a callable timer event function interface for the developers. You need to make your function based on this interface and you can simply pass it as parameter at the timer creation.

```
typedef void __timer_function(int pid);
```

Create timer entity

You can create a timer entity using the calling below. The created timer entities are stored centrally, you have access only to the id you provides at the creation.

```
int timer = create_timer(timer_id, period, timer_function);
```

function parameters:

0 – **[int]** you need to provide an id for the timer (it will be the return value of the calling as well)

1 - **[int]** period value (usually milliseconds)

2 - **[__p_timer_function]** your timer function based on **__timer_function** interface

You can drop the timer entity with the following statement:

```
drop_timer(timer_id);
```

function parameters:

0 – **[int]** the id of the timer entity

Timer pool

Timer pool is a cycle, which calls your created timer entities. You can start this cycle with the following calling, don't forget, that it runs on the caller thread.

timer_pool();

You can exit from the timer pool cycle calling the timer_pool_exit function, of course, one of your timer entities need to do it!

timer_pool_exit();

String support

String handler functions are available.

Known issues

In case of foreign key based field there is no parent record checking at update and insert.

Hard to combine cursor loop and navigation with insert and delete.