2022

Description of the libfmrt library (v.1.0)

ROBERTO MAMELI

Index

Introduction

libfmrt is a library written in C language, which can be statically or dynamically linked to any C/C++ source code, providing Fast handling of Memory Resident Tables (hence the meaning of the acronym *fmrt*). It offers the ability to manage huge amount of data stored into in-memory tables, giving high efficiency and $\boldsymbol{O(\log_2 N)}$ complexity. To achieve this requirement, in-memory data are stored into tables organized as AVL trees (see ref. 1 in chapter References).

libfmrt is not a Data Base. As such, it does not fit tasks that requires some typical DB properties, such as persistence or capability to perform SQL queries or to join tables. Rather, it is suitable for tasks that require optimized handling of volatile data, such as implementation of a fast in-memory cache or handling of great amount of temporary data.

Main *libfmrt* characteristics

The ***libfmrt*** library provides the following characteristics:

- In-memory handling of up to ***32*** tables, where each table is characterized by:
  - A primary key and a variable number of attributes (from ***0*** to ***16***);
  - Key and attributes may be either:
    - 32-bit unsigned integers (in the interval between ***0*** and $2^{32}-1$);
    - 32-bit signed integers (in the interval between -2^{31} and $2^{31}-1$);
    - double precision floating point numbers;
    - characters;
    - strings (max length ***255*** characters);
    - timestamps
  - A single table can contain up to 2^{26} elements (i.e. slightly more than ***67*** millions, exactly ***67.108.864***)

- All tables are handled in-memory:
  - Previous limits are theoretical, actual limits depend upon memory availability in the system
  - The library uses memory from the heap space available in the system
  - In most systems, there are generally no restrictions on the heap size, other than the physical memory size

- High efficiency:
  - Being stored in memory, the tables benefit from available high speed and random-access capabilities
  - Moreover, all tables are stored as AVL trees (see ref. 1 in chapter References)
  - Worst case complexity of all read and write operations is $O(\log_2 N)$

- Library is thread-safe

- Available methods are listed below (see *libfmrt* API Description for detailed explanation):
  - *fmrtDefineTable()*
  - *fmrtClearTable()*
  - *fmrtDefineKey()*
  - *fmrtDefineFields()*
  - *fmrtRead()*
  - *fmrtCreate()*
  - *fmrtModify()*
  - *fmrtCreateModify()*
  - *fmrtDelete()*
  - *fmrtImportTableCsv()*
  - *fmrtExportTableCsv()*
  - *fmrtExportRangeCsv()*
  - *fmrtCountEntries()*
  - *fmrtGetMemoryFootPrint()*
  - *fmrtDefineTimeFormat()*
  - *fmrtEncodeTimeStamp()*
  - *fmrtDecodeTimeStamp()*

How to compile the *libfmrt* library

The library has been developed in Linux environment (***RedHat***, ***CentOS***), but since it relies on ***POSIX*** standards and ***gcc*** compiler and development toolkit, it can be easily ported to most UNIX based operating systems (just recompiling it). It can be compiled without problems with ***glibc 2.12*** or above.

Be aware that this library does not come with an automatically generated ***makefile*** (***cmake*** or similar). It contains a manually written ***makefile***, composed by a few lines, that works on ***RedHat*** based operating systems (***RedHat***, ***Centos***, etc.), and that can be easily adapted to other Linux based operating systems.

After having downloaded the library, extract the source files into a directory arbitrarily chosen in your Linux Box:

```
tar -zxvf libfmrt1.0.tar.gz
cd libfmrt1.0
```

(this example assumes *libfmrt1.0*, however it can be applied also to other versions by simply referring to the correct one).

After that, type the following commands:

```
make all
make install
```

The first compiles the library and produces in the *libfmrt1.0* directory both the static and the shared libraries (respectively *libfmrt.a* and *libfmrt.so.1.0*).

The second installs the libraries into the destination folders. Specifically, the header file *fmrt.h* is copied into */usr/local/include* (this path is the one usually used in *RedHat* based operating systems, it may differ in other Linux distributions).

Static library *libfmrt.a* is copied into the *../lib* folder. Dynamic libraries, instead, are copied to */usr/local/lib* path (usually used in *RedHat* based operating systems, it may differ in other Linux distributions).

Be aware that to use shared libraries, this path shall be either configured in */etc/ld.so.conf* or in environment variable *$LD_LIBRARY_PATH*. The first time you install the libraries, a further command might be needed to configure dynamic linker run-time bindings:

```
ldconfig
```

If you apply changes to the library source code and you want to recompile it from scratch, you can clean up all executables by typing:

```
make clean
```

How to use the *libfmrt* library into your C/C++ code

Let's assume that libraries are correctly compiled and installed (see previous section).

To use library functions within C/C++ source code, the following shall be done:

- include the ***fmrt.h*** header file

```
#include "fmrt.h"
```

- link the executable by including either the shared or the static library ***libfmrt***

To compile a generic example file (let's say ***example.c***), simply type:

```
gcc -g -c -O2 -Wall -v –I/usr/local/include example.c
gcc -g -o example example.c -lfmrt
```

for shared library linking or:

```
gcc -static example.c -I/usr/local/include -L. -lfmrt -o example
```

for static linking. In the previous command ***-L .*** means that the ***libfmrt.a*** file is available in the same directory of the source code ***example.c***; if this is not the case just replace the dot after ***L*** with the path to the library file.

libfmrt API Description

The current ***libfmrt*** library (***v.1.0.0***) provides 17 methods:

- *fmrtDefineTable()*
- *fmrtClearTable()*
- *fmrtDefineKey()*
- *fmrtDefineFields()*
- *fmrtRead()*
- *fmrtCreate()*
- *fmrtModify()*
- *fmrtCreateModify()*
- *fmrtDelete()*
- *fmrtImportTableCsv()*
- *fmrtExportTableCsv()*
- *fmrtExportRangeCsv()*
- *fmrtCountEntries()*
- *fmrtGetMemoryFootPrint()*
- *fmrtEncodeTimeStamp()*
- *fmrtDecodeTimeStamp()*
- *fmrtDefineTimeFormat()*

All library functions listed above use a common set of constants and type definitions, which are described immediately below, while the following sections provide a detailed description of all library calls.

libfmrt Types and Constants

The following custom (simple) types are defined by the library:

- ***fmrtId***
  8-bit unsigned integer in the interval ***0***-***255***, it is used by ***libfmrt*** methods as identifier (e.g. table identifier)

- ***fmrtType***
  8-bit unsigned integer in the interval ***0***-***255***, it denotes a data type for key and fields of in-memory tables;

- ***fmrtLen***
  8-bit unsigned integer in the interval ***0***-***255***, it is used to represent field lengths;

- ***fmrtResult***
  8-bit unsigned integer in the interval ***0***-***255***, it provides the result code of library operations;

- ***fmrtParamMask***
  16-bit unsigned integer in the interval ***0***-***65535***, it is used as a bitmask by *fmrtModify()* and *fmrtCreateModify()* methods;

- ***fmrtIndex***
  32-bit unsigned integer used to express the number of elements in a table by *fmrtDefineTable()* and *fmrtCountEntries()* methods.

The ***fmrtType*** type is used to denote types for keys and fields. It can assume the following values:

- ***FMRTINT (0)***
  A key/field of this type is constituted by a 32-bit Unsigned Integer (between ***0*** and $2^{32}-1$)

- ***FMRTSIGNED (1)***
  This type represents a 32-bit Signed Integer (between -2^{31} and $2^{31}-1$)

- ***FMRTDOUBLE (2)***
  Key/fields defined as ***FMRTDOUBLE*** represent double precision floating point numbers

- ***FMRTCHAR (3)***
  A key/field of this type is constituted by an 8-bit character (ASCII code)

- ***FMRTSTRING (4)***
  This represents a null terminated string, with a maximum length of ***255*** characters

- ***FMRTTIMESTAMP (5)***
  A key/field of this type denotes a Time Stamp (i.e. Unix ***time_t*** type)

Variables defined as ***fmrtResult*** type can assume one of the following values:

- ***FMRTOK (0)***
  The operation has been completed successfully

- ***FMRTKO (1)***
  The operation failed due to an unspecified error

- ***FMRTIDALREADYEXISTS (2)***
  The Id in the request is already in use

- ***FMRTIDNOTFOUND (3)***
  The Id in the request does not exist

- ***FMRTMAXTABLEREACHED (4)***
  The maximum number of tables has been reached

- ***FMRTMAXFIELDSINVALID (5)***
  The number of fields is outside the allowed range

- ***FMRTREDEFPROHIBITED (6)***
  Key/Field Redefinition is not allowed

- ***FMRTDUPLICATEKEY (7)***
  The specified Key already exists in Create operation

- ***FMRTNOTEMPTY (8)***
  The Table contains at least one element

- ***FMRTNOTFOUND (9)***
  Searched Element has not been found

- ***FMRTFIELDTOOLONG (10)***
  String field exceeds max allowed length (***255*** characters)

- ***FMRTOUTOFMEMORY (11)***
  No more space left for new elements

The following constants are used in ***fmrtExportTableCSV()*** library call to specify the desired ordering of the exported file:

- ***FMRTASCENDING (0)***
  The CSV file shall be exported in ascending order with respect to the key

- ***FMRTDESCENDING (1)***
  The CSV file shall be exported in descending order with respect to the key

- ***FMRTOPTIMIZED (2)***
  The CSV file shall be exported using an order that minimizes the time needed for data reload

Finally, there is a constant that might be obtained as return value of ***fmrtCountEntries()***:

- ***FMRTNULLPTR***
  It is used by variables defined as ***fmrtIndex*** to denote an invalid index.

libfmrt Detailed Library Calls Description

fmrtDefineTable()

Function Prototype

```
fmrtResult fmrtDefineTable (fmrtId tableId, char* tableName, fmrtIndex tableNumElem)
```

Parameters

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255***; the library does not allow the definition of two tables with the same identifier.
- ***tableName (type char *)***
  Null terminated string (up to ***32*** characters) which identifies the table name. In case of length exceeding this limit, the name will be truncated.
- ***tableNumElem (type fmrtIndex)***
  Maximum number of elements for the table, it is a table attribute fixed at table definition, and cannot be changed later. It shall be included between ***1*** and 2^{26}.

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  Table successfully defined
- ***FMRTKO***
  Result obtained when the number of elements is outside the allowed interval
- ***FMRTIDALREADYEXISTS***
  The specified ***tableId*** is already in use by another table
- ***FMRTMAXTABLEREACHED***
  Result obtained when more than ***32*** tables are defined

Description

This library call is used to define a new table. This shall necessarily be the first library function invoked by the caller (a table cannot be used unless it is defined first). The library supports the definition of up to ***32*** tables.

A table cannot be redefined unless it is cleared first through ***fmrtClearTable()***.

Please observe that internal memory for the table is not reserved at this stage, rather it is allocated upon first element insertion.

fmrtClearTable()

Function Prototype

```
fmrtResult fmrtClearTable (fmrtId tableId)
```

Parameters

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  Table successfully deleted
- ***FMRTKO***
  Obtained if this function is invoked as first library call
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)

Description

This function is used to delete a table previously defined through ***fmrtDefineTable()***. If the memory for the table has already been allocated, it also deallocates this space: this happens only if at least one element has been inserted in the table, otherwise the call simply deletes table definition.

A ***tableId*** cannot be re-used unless ***fmrtClearTable()*** is invoked first.

This operation is destructive and cannot be recovered. If the table is not empty when the function is invoked, all its content is irreversibly lost.

fmrtDefineKey()

Function Prototype

```
fmrtResult fmrtDefineKey (fmrtId tableId, char* keyName, fmrtType keyType,
fmrtLen keyLen)
```

Parameters

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***keyName (type char *)***
  Descriptive name of the key (up to ***16*** characters allowed, in case of length exceeding this limit the name will be truncated).
- ***keyType (type fmrtType)***
  Identifies key type. Allowed values are defined in section ***libfmrt Types and Constants***.
- ***keyLen (type fmrtLen)***
  Meaningful only in case of ***FMRTSTRING keyType*** when it represents the maximum string length for the key value. Allowed values are in the interval ***1***-***255***. For other key types, the parameter is meaningless.

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  Key definition successful
- ***FMRTKO***
  Result obtained either when this is the first library call invoked by the caller or when ***keyType*** parameter is not valid
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)
- ***FMRTREDEFPROHIBITED***
  The key has already been defined and cannot be redefined
- ***FMRTFIELDTOOLONG***
  In case of ***FMRTSTRING keyType***, the ***keyLen*** parameter is outside the allowed interval

Description

This call is used to define the key name, type (and also key length for string key type) for a previously defined Table.

This call can be invoked only once after ***fmrtDefineTable()*** and before invoking ***fmrtDefineFields()***. Multiple invocations are forbidden.

fmrtDefineFields()

Function Prototype

```
fmrtResult fmrtDefineFields (fmrtId tableId, uint8_t numFields, ...)
```

Parameters

This call has a variable number of arguments. The first two parameters (mandatory) are respectively:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***numfields (type uint8_t)***
  Number of fields for each table row, in the interval ***1***-***16*** (excluding the key, which is defined apart through ***fmrtDefineKey()***).

The parameters after the two mentioned above are used to specify name, type (and length for string field type) for each field of the table. Specifically, for each field, there shall be:

- ***fieldName (type char *)***
  Descriptive name of the field (up to ***16*** characters allowed, in case of length exceeding this limit the name will be truncated).
- ***fieldType (type fmrtType)***
  Identifies field type. Allowed values are defined in section ***libfmrt Types and Constants***.
- ***fieldLen (type fmrtLen)***
  This shall be inserted only if field type = ***FMRTSTRING***. Please observe that the function behaviour is unpredictable in case parameter is inserted for other field types (e.g. ***FMRTINT*** or ***FMRTCHAR***). In case of ***FMRTSTRING fieldType*** it represents the maximum string length for the field value. Allowed values are in the interval ***1***-***255***.

The number and type of parameters specified in the library call shall match the number of fields defined by the ***numfields*** parameter, otherwise the function behaviour is unpredictable.

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  Field definition successful
- ***FMRTKO***
  Result obtained either when this is the first library call invoked by the caller or when one of the ***fieldType*** parameters is not valid
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)
- ***FMRTREDEFPROHIBITED***
  Fields have already been defined and cannot be redefined
- ***FMRTMAXFIELDSINVALID***
  The specified number of fields is outside the allowed range (***1***-***16***)
- ***FMRTFIELDTOOLONG***
  In case of ***FMRTSTRING keyType***, the ***keyLen*** parameter is outside the allowed interval

Description

This call is used to define the name, type (and also length for string type) of the fields for a previously defined Table. It is not mandatory (i.e. a Table may not contain any field, but just the key).

This call can be invoked only once after ***fmrtDefineKey()*** and before invoking ***fmrtCreate()***. Multiple invocations are forbidden.

fmrtRead()

Function Prototype

```
fmrtResult fmrtRead (fmrtId tableId, ...)
```

Parameters

This call has a variable number of arguments. The first two parameters (mandatory) are respectively:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***key (type previously specified through fmrtDefineKey())***
  Contains the key value to be searched into the table. It shall be of the same type defined by the ***fmrtDefineKey()*** call. The library behaviour is undefined if this constraint is not satisfied. In case of string key, the length will be truncated to the max length specified at key definition through ***fmrtDefineKey()***).

After the two mandatory parameters mentioned above, there is a list of pointer parameters that are filled with values read from the table entry (if present). Parameters are ordered according to the same order used in definition (i.e. in ***fmrtDefineFields()*** library call). Be aware that parameters number and types shall be correct, otherwise the call may cause run-time errors.

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  The entry has been found and corresponding parameters extracted. The fields retrieved from the table are stored in pointers passed as arguments as already explained above.
- ***FMRTNOTFOUND***
  The entry with the given key is not present in the table
- ***FMRTKO***
  Result obtained when this is the first library call invoked by the caller
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)

Description

This library call is used to read an entry from the table. It requires as input parameters the ***tableId*** and the searched key. If the key is present, the remaining parameters are filled with values extracted from the table. Otherwise, a proper error code is returned.

fmrtCreate()

Function Prototype

```
fmrtResult fmrtCreate (fmrtId tableId, ...)
```

Parameters

This call has a variable number of arguments. The first two parameters (mandatory) are respectively:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***key (type previously specified through fmrtDefineKey())***
  Contains the key value of the new entry which is being inserted into the table. It shall be of the same type defined by the ***fmrtDefineKey()*** call. The library behaviour is undefined if this constraint is not satisfied. In case of string key, the length will be truncated to the max length specified at key definition through ***fmrtDefineKey()***).

After the two mandatory parameters mentioned above, there is a list of arguments that are used to fill the fields of the new entry that is going to be created. Parameters are ordered according to the same order used in definition (i.e. in ***fmrtDefineFields()*** library call). Be aware that parameters number and types shall be correct, otherwise the call may cause run-time errors. In case of string parameters, the length will be truncated to the max length specified at field definition through ***fmrtDefineFields()***.

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  The entry has been correctly inserted into the table.
- ***FMRTDUPLICATEKEY***
  The entry has not been inserted since the given key is already present in the table (***libfmrt*** requires uniqueness of the key)
- ***FMRTKO***
  Result obtained when this is the first library call invoked by the caller
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)
- ***FMRTOUTOFMEMORY***
  The entry has not been inserted since the table is full.

Description

This library call is used to insert a new entry into the table whose identifier is specified as first parameter (***tableId***). Since ***libfmrt*** mandates key uniqueness, the operation is allowed only if an entry with the same key is not already present, and if there is still available space in the table, otherwise, a proper error code is returned.

fmrtModify()

Function Prototype

```
fmrtResult fmrtModify (fmrtId tableId, fmrtParamMask paramMask, ...)
```

Parameters

This call has a variable number of arguments. The first three parameters (mandatory) are respectively:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***paramMask (type fmrtParamMask)***
  It is a bitwise mask that is used to identify the parameters to be changed. See Description below for further details about parameter's usage and meaning.
- ***key (type previously specified through fmrtDefineKey())***
  Contains the key value of the entry which shall be modified. It shall be of the same type defined by the ***fmrtDefineKey()*** call. The library behaviour is undefined if this constraint is not satisfied. In case of string key, the length will be truncated to the max length specified at key definition through ***fmrtDefineKey()***).

After those mandatory parameters, there is a list of arguments that are used to update the fields of the entry that is going to be modified. Parameters are ordered according to the same order used in definition (i.e. in ***fmrtDefineFields()*** library call). Be aware that parameters number and types shall be correct, otherwise the call may cause run-time errors.

All fields shall be included in the call, but the only ones that will be updated are those pointed by the ***paramMask***, as explained below. In case of string parameters, the length will be truncated to the max length specified at field definition through ***fmrtDefineFields()***.

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  The entry has been correctly updated into the table
- ***FMRTNOTFOUND***
  The entry with the given key is not present in the table
- ***FMRTKO***
  Result obtained when this is the first library call invoked by the caller
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)

Description

This library call is used to update an existing entry into the table whose identifier is specified as first parameter (***tableId***). The call does not insert a new entry in case it is not already present (if so, a proper error is obtained).

All fields shall be specified into the call, paying attention that number and type of the corresponding parameters shall fit the definition given through ***fmrtDefineFields()***, in order to avoid runt-time errors and/or unpredictable behaviour.

Be aware that only fields enabled through the ***paramMask*** specified as second parameter will be updated, the remaining ones will not be changed. In the ***paramMask*** parameter, the rightmost bit represents the first parameter in ***fmrtDefineFields()***, the one immediately at its left is the second parameter, and so on. For example, assuming that 5 parameters have been defined through ***fmrtDefineFields()***, e.g.:

```
fmrtDefineFields(tableId,5, “param1”,..,”param2”,..,”param5”,..)
```

and that we want to update only parameters ***param1***, ***param2*** and ***param5***, we would set ***paramMask*** as follows:

```
paramMask = 19    →    binary 10011
```

fmrtCreateModify()

Function Prototype

```
fmrtResult fmrtCreateModify (fmrtId tableId, fmrtParamMask paramMask, ...)
```

Parameters

This call is quite similar to ***fmrtModify()***, with the notable difference that it either modifies a table entry (if already present) or insert it as a new entry otherwise. It has a variable number of arguments. The first three parameters (mandatory) are respectively:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***paramMask (type fmrtParamMask)***
  It is a bitwise mask that is used to identify the parameters to be changed (in case update of an existing entry). See Description below for further details about parameter's usage and meaning. This parameter is meaningless in case of insertion of new entry.
- ***key (type previously specified through fmrtDefineKey())***
  Contains the key value of the entry which shall be modified. It shall be of the same type defined by the ***fmrtDefineKey()*** call. The library behaviour is undefined if this constraint is not satisfied. In case of string key, the length will be truncated to the max length specified at key definition through ***fmrtDefineKey()***).

After those mandatory parameters, there is a list of arguments used to set field values of the entry that is going to be inserted/modified. Parameters are ordered according to the same order used in definition (i.e. in ***fmrtDefineFields()*** library call). Be aware that parameters number and types shall be correct, otherwise the call may cause run-time errors.
In any case, all parameters shall be included in the call. In case of update of an existing entry, the only fields that will be updated are those pointed by the ***paramMask***, as explained below. If the specified key is not present in the table, a new entry will be added. In this case, all parameters passed as arguments will be used to set values of the new entry's fields, independently from the ***paramMask***.
As usual, in case of string parameters, the length will be truncated to the max length specified at field definition through ***fmrtDefineFields()***.

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  The entry has been correctly inserted or updated into the table
- ***FMRTKO***
  Result obtained when this is the first library call invoked by the caller
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)
- ***FMRTOUTOFMEMORY***
  The entry has not been inserted since the table is full.

Description

This library call is used to update an existing entry into the table whose identifier is specified as first parameter (***tableId***). Differently from ***fmrtModify()***, it does not provide an error in case of

entry not found into the table, rather it inserts a new entry by setting all field values independently from the ***paramMask*** specified as second argument.

All fields shall be specified into the call, paying attention that number and type of the corresponding parameters shall fit the definition given through ***fmrtDefineFields()***, in order to avoid runt-time errors and/or unpredictable behaviour.

Be aware that, in case of update of an existing entry, only fields enabled through the ***paramMask*** specified as second parameter will be changed, the remaining ones will not be affected. In the ***paramMask*** parameter, the rightmost bit represents the first parameter in ***fmrtDefineFields()***, the one immediately at its left is the second parameter, and so on. For example, assuming that 5 parameters have been defined through ***fmrtDefineFields()***, e.g.:

```
fmrtDefineFields(tableId,5, param1,..,param2,..param5)
```

and that we want to update only parameters ***param1***, ***param2*** and ***param5***, we would set ***paramMask*** as follows:

```
paramMask = 19    →    binary 10011
```

In case of insertion of a new entry, the ***paramMask*** will be ignored and all fields will be set according to parameters passed as arguments.

fmrtDelete()

Function Prototype

```
fmrtResult fmrtDelete (fmrtId tableId, ...)
```

Parameters

This call has two parameters (mandatory), which are respectively:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***key (type previously specified through fmrtDefineKey())***
  Contains the key value to be deleted from the table. It shall be of the same type defined by the ***fmrtDefineKey()*** call. The library behaviour is undefined if this constraint is not satisfied. In case of string key, the length will be truncated to the max length specified at key definition through ***fmrtDefineKey()***).

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  The entry has been found and successfully deleted from the table.
- ***FMRTNOTFOUND***
  The entry with the given key is not present in the table
- ***FMRTKO***
  Result obtained when this is the first library call invoked by the caller
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)

Description

This library call is used to delete an entry from the table. It requires as input parameters the ***tableId*** and the searched key. If the key is present, the corresponding entry is deleted from the table. Otherwise, a proper error code is returned.

fmrtImportTableCsv()

Function Prototype

```
fmrtResult fmrtImportTableCsv (fmrtId tableId, FILE *filePtr, char separator, int *lines)
```

Parameters

This call has the following parameters:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***filePtr (type FILE *)***
  It is a pointer to a file, which shall be opened in read mode by the caller before invoking this call. It cannot be NULL, otherwise an error will be provided.
- ***separator (char)***
  It is a char specified by the caller that is recognized as a separator between consecutive fields into the input CSV file.
- ***lines (int *)***
  It is a pointer to an integer parameter that is provided back to the caller. It contains either the total number of lines read from the input CSV file (in case of successful outcome) or the line number affected by the error (in case of error).

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  The table has been successfully imported from the input CSV file. In this case, the last parameter is set to the total number of lines read from the file
- ***FMRTKO***
  This is the result obtained when this is the first library call invoked by the caller or when the specified file pointer is NULL. In those cases, the last parameter provides 0.
  This result code is also obtained when an error is detected while reading input lines from the file (e.g. incomplete line); in such cases the last parameter reports the line where the error has been detected. Elements read from the file up to the wrong line are inserted into the table.
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)
- ***FMRTOUTOFMEMORY***
  The maximum number of elements has been reached while inserting data into the table from the input CSV file (the maximum number of elements is specified at table definition as a parameter of ***fmrtDefineTable()***). The last parameter identifies the line in the input file where data import was stopped.

Description

This library call is used to import the content of a given file in CSV format (specified by the file pointer given by the second parameter) into the table whose id is provided by the first parameter. The separator used by the input file is specified by the third parameter.

Please note that the file shall be opened before calling this function, otherwise a run-time error will occur.

Similarly, the function call does not close the input file, which must be closed by the caller.

Data read from the file are appended to existing data in the table (if the input table is not empty).

In case of duplicate key, the existing entry is overwritten: no errors are provided in this case.

Please, be aware that the time needed to import a CSV file depends on several factors: number of elements in the file, length of the input lines and their order.
Concerning the number of elements, this is quite intuitive: the greater the number of input lines, the longer the time needed to import them. Please be aware that on the average, the import time increases more than linearly with the number of elements.
Moreover, a file of 100,000 rows made up by 3-4 numeric fields is loaded definitely faster than a file made by the same number of rows, each one consisting of several (quite) long strings.
Finally, even elements ordering matters. The worst case occurs when elements in the input file are correctly ordered (either in ascending or in descending order). In fact, in this case the internal data structures shall be properly re-organized upon each insertion (i.e. the internal binary tree shall be properly rebalanced). The ***fmrtExportTableCsv()*** function described below provides the possibility to export data according to an optimized order, that can be reloaded through ***fmrtImportTableCsv()*** without the need for rebalancing, with a decrease of the reload time around 10%.

fmrtExportTableCsv()

Function Prototype

```
fmrtResult fmrtExportTableCsv (fmrtId tableId, FILE *filePtr, char separator, uint8_t selectedOrder)
```

Parameters

This call has the following parameters:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***filePtr (type FILE *)***
  It is a pointer to a file, which shall be opened in write (or append) mode by the caller before invoking this call. If NULL, the output is written on ***stdout***.
- ***separator (char)***
  It is a char specified by the caller that is used to separate fields into the output.
- ***selectedOrder (uint8_t)***
  This parameter can assume one of the following values: ***FMRTASCENDING***, to export data in ascending order with respect to the key, ***FMRTDESCENDING*** to use descending order, and ***FMRTOPTIMIZED*** to export data using an order optimized to speed up data reload through ***fmrtImportTableCsv()***. If an unrecognized value is specified, by default the call will export data assuming ***FMRTOPTIMIZED***.

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  The table has been successfully exported to the output CSV file.
- ***FMRTKO***
  This is the result obtained when this is the first library call invoked by the caller.
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)
- ***FMRTOUTOFMEMORY***
  There is not enough memory available in the system to allow the allocation of internal data structures needed to export data according to ***FMRTOPTIMIZED*** ordering.

Description

This library call is used to export the content of a table (whose ***tableId*** is specified as first parameter) into a given file in CSV format (pointed by the file pointer given by the second parameter). The separator used to separate fields in the output file is specified by the third parameter. The fourth parameter is used to specify the desired data export ordering.

Please note that the file shall be opened before calling this function, otherwise a run-time error will occur.

Similarly, the function call does not close the output file, which must be closed by the caller.

fmrtExportRangeCsv()

Function Prototype

```
fmrtResult fmrtExportRangeCsv (fmrtId tableId, FILE *filePtr, char separator, uint8_t selectedOrder, ...)
```

Parameters

This call has the following parameters:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).
- ***filePtr (type FILE *)***
  It is a pointer to a file, which shall be opened in write (or append) mode by the caller before invoking this call. If NULL, the output is written on ***stdout***.
- ***separator (char)***
  It is a char specified by the caller that is used to separate fields into the output.
- ***selectedOrder (uint8_t)***
  This parameter can assume one of the following values: ***FMRTASCENDING***, to export data in ascending order with respect to the key or ***FMRTDESCENDING*** to use descending order. Differently from ***fmrtExportTableCsv()***, ***FMRTOPTIMIZED*** is not supported. If an unrecognized value is specified, by default the call will export data assuming ***FMRTASCENDING***.
- ***keyMin (type previously specified through fmrtDefineKey())***
  It contains the minimum key value that delimits the lower bound of the export interval. It shall be of the same type defined by the ***fmrtDefineKey()*** call. The library behaviour is undefined if this constraint is not satisfied. In case of string key, the length will be truncated to the max length specified at key definition through ***fmrtDefineKey()***).
- ***keyMax (type previously specified through fmrtDefineKey())***
  It contains the maximum key value that delimits the upper bound of the export interval. It shall be of the same type defined by the ***fmrtDefineKey()*** call. The library behaviour is undefined if this constraint is not satisfied. In case of string key, the length will be truncated to the max length specified at key definition through ***fmrtDefineKey()***).

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  The table has been successfully exported to the output CSV file.
- ***FMRTKO***
  This is the result obtained when this is the first library call invoked by the caller.
- ***FMRTIDNOTFOUND***
  tableId does not represent a valid table (i.e. a table previously defined by ***fmrtDefineTable()***)

Description

This library call is used to export partially the content of a table (whose ***tableId*** is specified as first parameter) into a given file in CSV format (pointed by the file pointer given by the second parameter). The separator used to separate fields in the output file is specified by the third parameter. Only the entries with key in the interval between ***keyMin*** and ***keyMax*** are exported.

Please note that the file shall be opened before calling this function, otherwise a run-time error will occur.

Similarly, the function call does not close the output file, which must be closed by the caller.

fmrtCountEntries()

Function Prototype

```
fmrtIndex fmrtCountEntries(fmrtId tableId)
```

Parameters

This call has just one input parameter:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).

Return Value

The library call provides the number of elements stored into the table, or ***FMRTNULLPTR*** in case of any error (e.g. table not defined).

Description

See above.

fmrtGetMemoryFootPrint()

Function Prototype

```
long fmrtGetMemoryFootPrint(fmrtId tableId)
```

Parameters

This call has just one input parameter:

- ***tableId (type fmrtId)***
  It's the unique identifier of the table between ***0*** and ***255*** specified at table definition (***fmrtDefineTable()***).

Return Value

The library call provides the memory occupancy of the table expressed in bytes, or ***0*** in case of any problem (e.g. table not defined).

Description

See above.

fmrtDefineTimeFormat()

Function Prototype

```
fmrtResult fmrtDefineTimeFormat(char * fmrtFormat)
```

Parameters

This call takes just one input parameter:

- ***fmrtFormat (type char *)***
  This is a string representing the desired time format, according to the same convention specified by ***strftime()*** man page (e.g. ***"%c"*** for full date and time, ***"%T"*** for time in ***HH:MM:YY*** format, ***"%F"*** for date in the format ***YYYY-MM-DD***, etc.).
  In case the parameter is either ***NULL*** or an empty string, then time format is set to raw (i.e. timestamps are represented as raw ***time_t*** data)

Return Value

The library call provides one of the following ***fmrtResult*** values:

- ***FMRTOK***
  Time format correctly defined and set.
- ***FMRTKO***
  Time format not accepted (e.g. due to wrong ***fmrtFormat*** parameter).

Description

This library call allows to change timestamp representation of ***FMRTTIMEFORMAT*** data handled by the library.

Please observe that the library manages timestamps internally using always the Linux representation (i.e. seconds from ***Epoch***, ***Jan 1st 1970, 00:00:00 UTC***), however it allows changing the format used to display and enter ***FMRTTIMEFORMAT*** data. When a format is specified, then timestamp keys and fields in ***fmrtRead()***, ***fmrtCreate()***, etc. must be expressed as character strings with the corresponding format (e.g. "***Mon Jun 21 17:08:55 2021***").

The library allows also the usage of raw time format; in that case, timestamp keys and fields in ***fmrtXxxx()*** routines must be defined as ***time_t*** data.

This routine is OPTIONAL. If not called, default time format will be set to "***%c***", i.e. current locale.

The routine can be called as many times as desired. This will not affect internal data representation, but only data management and display (e.g. the format of timestamp data in CSV output files).

Finally, observe that timestamp format is GLOBAL, i.e. it is a configuration that affects all tables.

fmrtEncodeTimeStamp()

Function Prototype

```
time_t fmrtEncodeTimeStamp(char * timeStamp)
```

Parameters

This call takes just one input parameter:

- ***timeStamp (type char *)***
  Input string that represents a time stamp. It shall be formatted according to the timestamp format defined through ***fmrtDefineTimeFormat()***.

Return Value

This function provides the timestamp encoded as a ***time_t*** data or ***0*** in case of errors.

Description

This library call takes an input string representing a time stamp, properly expressed according to the configured format, and provides back the corresponding value encoded as a ***time_t*** (Linux time).

fmrtDecodeTimeStamp()

Function Prototype

```
void fmrtDecodeTimeStamp(time_t rawTimeStamp, char * formattedTimeStamp)
```

Parameters

This call takes two input parameters:

- ***rawTimeStamp (type time_t)***
  Input value of the time stamp expressed as raw data (i.e. according to Linux ***time_t*** type).
- ***formattedTimeStamp (type char *)***
  Output string that contains the timestamp formatted according to the format defined through ***fmrtDefineTimeFormat()***.

Return Value

This function does not have return values. On return, it provides in the second parameter the timestamp properly formatted.

Description

This library call takes a raw time stamp (i.e. a ***time_t*** data) as first parameter and provides back in the second parameter a string formatted according to the timestamp format defined through ***fmrtDefineTimeFormat()***.

The string must be allocated before calling this function

Examples

The ***./examples*** subdirectory in the ***libfmrt*** package contains several examples that make use of the library for different purposes. To use them, it is assumed that libraries are correctly installed into ***/usr/local/lib*** and that this path is either configured in ***/etc/ld.so.conf*** or in environment variable ***$LD_LIBRARY_PATH***.

To compile a specific example file (let's say ***DictionaryWords.c***), simply type the following commands from the ***./examples/src*** subdirectory:

```
gcc -g -c -O2 -Wall -v -I../../headers DictionaryWords.c
gcc -g -o ../bin/DictionaryWords DictionaryWords.o -lfmrt
```

(for shared library linking) or:

```
gcc  ./DictionaryWords.c -I../../headers -L../../lib  -o ../bin/
DictionaryWords -lfmrt
```

However, the examples can be compiled all at once by typing either:

```
make static
```

or

```
make shared
```

Both commands above shall be issued from the ***./example*** subdirectory, and will produce executables in the ***./examples/bin*** subdirectory; specifically, the former will produce executables linked statically with the ***libfmrt*** library, while the latter will provide executables linked dynamically.

The following command clears all executables:

```
make clean
```

The ***./examples/data*** subdirectory contains some sample data useful to run examples.

The following sub-section provides a detailed description of all the examples provided with the library.

DictionaryWords

This example (***./examples/src/DictionaryWords.c***) defines a single table, called "***DictionaryWords***" of 80,000 words defined as 32-char strings. The word represents the key, and no additional fields are defined in the table.

When launched, the application prompts the following menu:

```
**********************
* Available choices *
**********************

        Menu
        ----

        (1) - Import words from file
        (2) - Search a word
        (3) - Insert a word in the table
        (4) - Delete a word from the table
        (5) - Count words
        (6) - Export all words to file in ascending order
        (7) - Export all words to file in optimized order
        (8) - Export range of words in ascending order
        (0) - Exit

        Enter the selected choice:
```

The choices are straightforward, there is no need to provide further explanations. Words can be read from an input file (at the purpose, directory ***./examples/data/dict*** contains a list of about 60,000 Italian words). After that, they can be searched, deleted, inserted or exported to a file.
The source code illustrates the usage of several ***libfmrt*** calls.

CountWordsOccurrence

The example application ***./examples/src/CountWordsOccurrence.c*** uses a single table, called "***WordCount***" of 120,000 elements, with a key constituted by a 32-char string ("***Word***") and a single integer field called "***Frequency***".

When launched, the application prompts the following menu:

```
**********************
* Available choices *
**********************

        Menu
        ----

        (1) - Count words from txt file
        (2) - Search a word and print number of occurrences
        (3) - Count distinct words
        (4) - Export all words to file in ascending order
        (5) - Export all words to file in optimized order
        (6) - Export range of words in ascending order
        (7) - Display Memory Footprint
        (0) - Exit

        Enter the selected choice:
```

Basically, the example application is able to scan an input text file and to count word occurrences. Words are identified by means of ***strtok()*** standard C function call. This is not completely correct; the algorithm should be refined a bit to isolate words in the correct way. In fact, this version recognizes things like "***1234***" or "***[[***" as words; moreover, it is case sensitive, therefore "***of***" and "***Of***" appear as distinct words. Nevertheless, I have decided not to spend too much time on refining this algorithm, since the purpose is just to illustrate the usage of ***libfmrt*** call, not to produce a sophisticated parser.

The ***./examples/data/books*** directory contains some English books formatted as text files[1] that can be used with option ***(1)*** for testing purposes:

```
I'm going to import and count words from an input txt file...
Please insert file name: ../data/books/Ivanhoe_Walter_Scott.txt
Finished reading txt file... 21560 lines read in 20 seconds


        Press the ENTER key to continue.
```

The remaining choices allow to search for a word and count the number of occurrences, to count the total number of distinct words, to export the table in CSV format, etc.:

[1] All English books have been downloaded from https://www.gutenberg.org/

```
Enter word to search:
        Word? of
Operation Succeeded
Word of is present and occurs 8147 times

        Press the ENTER key to continue...
```

```
The table contains 17048 distinct words

        Press the ENTER key to continue...
```

Option ***(7)*** provides the memory occupancy of the table. Given 120,000 elements made up by a 32-char string and a 4-byte unsigned integer, the memory footprint is about 5Mbytes:

```
The whole table occupies 5400480 bytes in the internal memory ...

        Press the ENTER key to continue...
```

Televoting

This sample program (***./examples/src/Televoting.c***) simulates televoting operations. When launched, it starts several threads, each one simulating the reception of a great number of preferences from random generated phone numbers. The number of concurrent threads, as well as the number of random votes generated per thread, are selected by the operator when the program is started:

```
This program launches several threads, each one simulating the reception
of a number of televotes from random generated telephone numbers.
Data are collected into a table (Votes), while events (e.g. duplicated
votes) are stored into another table (LoggedEvents).

Enter the number of threads (1-4)? 4
Enter the number of votes per thread (1-300000)? 25000

        Press the ENTER key to continue...
```

When ENTER is pressed, the program instantiates the specified number of threads:

```
Televoting in progress...

Thread (id cec44700) started (simulating 25000 votes)...
Thread (id d0447700) started (simulating 25000 votes)...
Thread (id cfc46700) started (simulating 25000 votes)...
Thread (id cf445700) started (simulating 25000 votes)...
```

Each thread contains a loop that generates random votes (between ***1*** and ***20***) for random generated telephone numbers (in the form ***+39301xxxxxx***). The votes are collected into a table, called ***Votes***, where the phone number constitutes the key and the expressed preference is the only attribute. Votes has a dimension of ***1,000,000*** elements.

In case of multiple votes from the same number, insertion into ***Votes*** table fails (remember that ***fmrtCreate()*** operation provides an error when the key is already present in the table). If this happens, an event is logged into another table (***LoggedEvents***), where the key is represented by the current timestamp and the only field is a string that reports the collision. ***LoggedEvents*** table has a maximum dimension of ***500,000*** elements.

As soon as the running threads complete their execution, a message is displayed by the program:

```
Thread (id cec44700) completed
Thread (id cfc46700) completed
Thread (id cf445700) completed
Thread (id d0447700) completed

Televoting operations ended... elapsed time 878 seconds


        Press the ENTER key to continue...
```

Pressing ENTER the following menu is displayed:

```
**********************
* Available choices *
**********************

        Menu
        ----

        (1) - Display Number of Elements, Size and Memory Footprint of Tables
        (2) - Search and Print Vote Expressed by Input Phone Number
        (3) - Export Preferences to File
        (4) - Export Events to File
        (0) - Exit

        Enter the selected choice:
```

Options are quite straightforward. Choices (2) allows to search for a given number and to see the corresponding vote (if any):

```
Enter phone number to search (e.g. +39301123456)? +39301051465
Operation Succeeded
Phone Number +39301051465 expressed the following vote: 9

        Press the ENTER key to continue...
```

Options (3) and (4) allows to export respectively the ***Votes*** and the ***LoggedEvents*** tables. The obtained files appear as follows:

```
#Table: Votes (Id: 4)
#PhoneNo,Preference
+39301000023,19
+39301000024,16
+39301000032,13
+39301000064,4
+39301000085,1
+39301000096,5
+39301000108,6
+39301000117,12
…
…
+39301468783,11
+39301468791,11
+39301468813,17
+39301468814,15
+39301468820,19
+39301468825,11
…
…
```

```
#Table: LoggedEvents (Id: 12)
#TimeStamp,Event
Thu Jun  9 14:58:32 2022,Multiple Occurrences of repeated votes
Thu Jun  9 14:58:33 2022,Multiple Occurrences of repeated votes
Thu Jun  9 14:58:34 2022,Multiple Occurrences of repeated votes
Thu Jun  9 14:58:35 2022,Multiple Occurrences of repeated votes
Thu Jun  9 14:58:36 2022,Multiple Occurrences of repeated votes
Thu Jun  9 14:58:37 2022,Multiple Occurrences of repeated votes
…
…
Thu Jun  9 15:07:47 2022,Multiple Occurrences of repeated votes
```

```
Thu Jun  9 15:07:48 2022,+39301956366 attempted to vote again
Thu Jun  9 15:07:49 2022,Multiple Occurrences of repeated votes
Thu Jun  9 15:07:50 2022,Multiple Occurrences of repeated votes
Thu Jun  9 15:07:51 2022,Multiple Occurrences of repeated votes
Thu Jun  9 15:07:52 2022,Multiple Occurrences of repeated votes
Thu Jun  9 15:07:53 2022,+39301225709 attempted to vote again
Thu Jun  9 15:07:54 2022,Multiple Occurrences of repeated votes
…
```

Finally, option (1) displays some useful information about the two managed tables: maximum number of elements, current number of elements and memory occupancy:

```
Table Votes:
        Table Id:         4
        Table Size:       1000000
        Number of Votes:  95277
        Memory Size (KB): 27344.24

Table LoggedEvents:
        Table Id:         12
        Table Size:       500000
        Number of Events: 869
        Memory Size (KB): 31738.77


        Press the ENTER key to continue...
```

This example illustrates the usage of several ***libfmrt*** functions (including the usage of timestamps as table key) and shows also concurrent access to tables from different threads.

BarCodeCache

This example is located in ***./examples/src/BarCodeCache.c***. It implements a memory cache that stores a set of barcodes (constituted by a 13-char fixed length string), along with a couple of associated attributes, representing respectively the size (24-char string) and the description (48-char string) of the associated product.

It uses a single table, called "***BarCodes***", of ***1,300,000*** elements.

At startup, the following menu appears:

```
**********************
* Available choices *
**********************

        Menu
        ----

        (1) - Import barcodes from CSV input file
        (2) - Search and Print Barcode
        (3) - Insert a new Barcode in the table
        (4) - Remove Barcode from the table
        (5) - Count Barcodes
        (6) - Export all Barcodes to file in optimized order
        (7) - Export range of Barcodes in ascending order
        (8) - Display Memory Footprint
        (0) - Exit

        Enter the selected choice:
```

Option (1) can be used to read data from a CSV file. The ***../data/largeDatasets*** subdirectory contains several example CSV files, of various sizes. Be aware that the time needed to load them increases more than linearly with respect to the number of elements. As an example, 1,000 elements are read almost instantaneously, 10,000 elements are read with a rate of about 1000 items/s, 100,000 elements are read with a rate of about 150 item/s, and so on. Ref. 1 in chapter References provides the rationale behind this behaviour: specifically, when elements are inserted in the right order, every insertion requires tree rebalancing. First elements are inserted very efficiently, but as the table grows, the rebalancing operation becomes more and more time consuming. Practically speaking, insertion begins very quickly, but it tends to slow down as the table fills up.

This is the output shown on the screen after selecting option (1) and providing a sample CSV file:

```
I'm going to import barcodes from an input file...
Please insert file name: ../data/largeDatasets/items_100K.csv
Finished reading input CSV file... 99981 lines read in 629 seconds

Operation Succeeded
Read 99981 lines from input file

        Press the ENTER key to continue...
```

The remaining choices allows to search, create or delete an input barcode (options (2), (3) and (4)), to count the number of elements (option (5)), to export data to a CSV file, using respectively the

ascending order (***FMRTASCENDING***, option (6)) or the optimized order (***FMRTOPTIMIZED***, option (7)) and, finally, to display the memory occupancy of the table (option (8)). The following pictures depicts the output obtained selecting options (2), (5) and (8) respectively:

```
13-char barcode to search? 3254560083622
Operation Succeeded
Barcode: 3254560083622 -> Size/Format: 250g | Description: Auchan Pur Arabica
Cafe Moulu

        Press the ENTER key to continue...
```

```
The table contains 99981 items

        Press the ENTER key to continue...
```

```
The whole table occupies 124800504 bytes in the internal memory ...

        Press the ENTER key to continue...
```

Despite the relevant table size (***1,300,000*** elements, each containing 3 string parameters for a total length of 13+24+48=***85*** characters), the memory occupancy is "only" 120Mbytes of internal memory.

Finally, in the following we provide an excerpt of the output CSV file obtained through option (6):

```
#Table: BarCodes (Id: 1)
#BarCode,Size/Format,Description
0880761618634,750 ml,CIROC LUXURY VODKA
0880761618702,1000 ml,CIROC LUXURY VODKA
0880761626240,750 ml,J W PURE MALT LGREEN LABELA
0880761626486,1750 ml,CIROC SNAP FROST VODKA
...
...
4937105059052,8.5 oz,Yuko System phiten repair treatment
4937182002248,117mm x 60mm - 185g,Avox Lighting - IL60-7.5w/760 (540lm) Natural
Sunlight LED - (100~240v)
4937182002255,117mmx60mm - 185g,Avox Lighting - 7.5w(460lm) Cool White LED -
(100~240v)
4937518227734,60 CAPSULES,3 SUPER SLIM BOMB
...
...
```

Known Issues and Future Improvements

This version of the ***libfmrt*** library is quite stable and is self-consistent enough to be released as a finished product. However, there are aspects that will certainly be the subject of improvement in future versions. This section provides a brief overview of the main issues that will be addressed in future releases.

- ***libfmrt v.1.0.0*** provides one library call for search operations (***fmrtRead()***), which basically allows only exact search of a given key. It does neither support search criteria other than equality, nor the usage of wildcards on string fields;

- this library version does not support search operation on fields other than the key;

- the library calls used to define the table structure (e.g. ***fmrtDefineFields()***) and to read and write data (***fmrtRead()***, ***fmrtCreate()***, etc.) are thought for applications in which the table structure is hardcoded in the application itself. However, imagine that an application needs to handle tables with structure defined at run time. With the current interface, this is simply impossible, since table structures (i.e. number, name and type of fields) shall be specified at coding time. To achieve this flexibility, a new interface with some new library calls shall be properly introduced.

- this version does not provide an efficient and safe way to save data on persistent storage. Indeed, it offers the possibility to save tables as CSV files (***fmrtExportTableCsv()***), but this might not be enough in some cases, at least for two reasons. First, data reload from a CSV input file is not particularly efficient (on lower-end hardware it might take hours for tables composed of some millions of elements); second, data exported in this way are easily readable by humans, and sometimes this might introduce confidentiality issues. Future versions of the library will be enhanced by introducing table binary dumps, able to address both observations above.

References

1. Introduction to AVL Trees, available at https://www.roberto-mameli.it/wp-content/uploads/2022/04/Tutorial_AVL_Trees.pdf