The **chronos** library

Contact: Brice Chardin
brice.chardin@liris.cnrs.fr

January 9, 2013

1 Introduction

This document gives an overview of the chronos API, and provides working examples for each access method. For convenience, these examples are also collected in a single compilable `example.cc` file in the `doc` directory, along with a simple `Makefile`.

For more information on chronos design, principles and performance, refer to [Chardin, 2011, chapter 4].

2 License

Chronos is free software. It may be used, free of charge, for any purpose, including commercial purposes. Chronos is distributed under the terms of the MIT License, reproduced here:

3 Chronos

Chronos is a software library providing an ordered key-value store. It is designed to efficiently manage historization data on flash memory, but is also able – possibly at the cost of performance – to manage many kind of key-value data on many kind of devices, including hard disk drives.

To use chronos for its data management, an application has to deal with: databases (cf. section 4), tables (cf. section 5), cursors (cf. section 6) and key-value pairs.

Key-value pairs are the fundamental data representation in chronos. These data elements are not typed: chronos manages only sequences of bytes. However, the application can (and should) establish a data structure to store multiple informations within a key or a value (cf. section 6 for an example).

Chronos is written in C++, compiled and tested on Linux distributions. However, no known limitation should prevent it to be compiled and run using other operating systems.

4 Chronos databases

A chronos database, represented by a `chr_chronos` object, is used to manage (create, drop, open and close) one or several tables on a single flash device (or file).

4.1 Initialization

A `chr_chronos` object is initialized as follows:

```
chr_chronos(const char * flash_device_path,
    const char * persistence_folder_path);
```

- `flash_device_path` is the path to the flash device. Chronos performs better on raw flash devices or partitions (which appear as special files, such as `/dev/sda2`) , but allows any file to be used to hold the database.
- `persistence_folder_path` is the path to a folder used by chronos to keep persistence-related files. These files are:
  - `chronos.chr`, which contains a description of the database with, for each table, its name (unique string identifier), the size of its keys, the size of its values and its B-tree root address.
  - `device.chr`, which contains the list of free sectors for this device.

For example, to open (or create) a chronos database on the device `/dev/sdb`, with persistence files `/var/chronos/chronos.chr` and `/var/chronos/device.chr`:

```
chr_chronos * chronos = new chr_chronos("/dev/sdb", "/var/chronos/");
```

Warning: if the device (in the example `/dev/sdb`) has a file system, every file it contains is lost.

4.2 Deinitialization

The persistence files are only written when the database is closed. The information maintained in these files can theoretically be rebuilt from the device's data, but this functionality is currently not implemented. It is therefore mandatory to properly close a database to retrieve its data on a subsequent access. To close a database, the `chr_chronos` object has to be destroyed:

```
delete chronos;
```

5 Chronos tables

A `chr_table_ctx` is a public wrapper for chronos table objects (`chr_table`). Chronos tables store fixed-size key-value pairs using B-trees.

Note: keys are sorted with a lexicographical comparator (`memcmp`). Consequently, integers have to be stored in big-endian form to be compared accurately.

5.1 Creation

A new table can be created in a database:

```
chr_chronos::create_table(const char * table_id, size_t size_of_keys,
    size_t size_of_values);
```

- `table_id` is a string used as a unique table identifier.
- `size_of_keys` and `size_of_values` are the fixed size (in bytes) of keys and values for this table.

For example, to create a table named `samples`, whose key is the concatenation of a 32 bit integer and a 64 bit integer, and value is a double:

```
chronos->create_table("samples", sizeof(uint32_t) + sizeof(uint64_t),
    sizeof(double));
```

An SQL equivalent to create such a table would be:

```
CREATE TABLE samples (
    sensor_id UNSIGNED INT NOT NULL,
    timestamp UNSIGNED BIGINT NOT NULL,
    value DOUBLE PRECISION NOT NULL,
    PRIMARY KEY (sensor_id, timestamp)
)
```

5.2 Deletion

A table can be dropped, deleting all its content and reclaiming space on the device.

Note: table contents are not immediately erased when a table is dropped: its allocated sectors are simply allowed to be overwritten, which will happen eventually.

```
chr_chronos::drop_table(const char * table_id);
```

- `table_id` is a string used as a unique table identifier.

For example, to drop the table `samples`:

```
chronos->drop_table("samples");
```

5.3 Initialization

To manage data in a table, it first has to be opened (and subsequently closed to correctly release resources).

Note: a table can be opened multiple times, but each context has to be closed.

```
chr_table_ctx * chr_chronos::open_table(const char * table_id);
```

- `table_id` is a string used as a unique table identifier.

For example, to open the table `samples`:

```
chr_table_ctx * samples_ctx = chronos->open_table("samples");
```

5.4 Deinitialization

The table context created with `open_table` has to be eventually closed to flush the B-tree root on disk and later correctly update the table's informations in the persistence file `chronos.chr`.

```
chr_chronos::close_table(chr_table_ctx * ctx);
```

For example:

```
chronos->close_table(samples_ctx);
```

6 Chronos cursors

In chronos, data is managed using cursors. A table context can be used to open and close such cursors.

6.1 Initialization

Multiple cursors can be opened concurrently for one or several tables. To open a cursor:

```
chr_table_ctx::open_cursor(const unsigned char * key, chr_pos_type type);
```

- `type` can be:
  - `CHR_KEY` to open a cursor on a specific key,
  - `CHR_FIRST` to open a cursor on the first key of the table (`key` should be `NULL`),
  - `CHR_LAST` to open a cursor on the last key of the table (`key` should be `NULL`).

For example, to open a cursor on sensor_id=10 and timestamp="2008-10-21 14:15:16.123" (converted by the application in 1224598516123, as a unix timestamp with milliseconds):

```
unsigned char * key = (unsigned char *) malloc(sizeof(uint32_t) +
    sizeof(uint64_t));
/* assign values to key, in big-endian form */
*((uint32_t*) key) = htobe32(10);
*((uint64_t*) key+sizeof(uint32_t)) = htobe64(1224598516123);
chr_cursor * cur = samples_ctx->open_cursor(key, CHR_KEY);
```

Note: it can be useful to define utilities – such as preprocessor macros – to access key and value elements individually.

```
#define SENSOR_ID(key) (*((uint32_t*) ((unsigned char*)(key))))
#define TIMESTAMP(key) (*((uint64_t*) (((unsigned char*)(key))
    + sizeof(uint32_t))))
#define VALUE(value) (*((double*) ((unsigned char*)(value))))

unsigned char * key = (unsigned char *) malloc(sizeof(uint32_t) +
    sizeof(uint64_t));
SENSOR_ID(key) = htobe32(10);
TIMESTAMP(key) = htobe64(1224598516123);
chr_cursor * cur = samples_ctx->open_cursor(key, CHR_KEY);
```

6.2 Deinitialization

A cursor has to be closed to release resources:

```
chr_table_ctx::close_cursor(chr_cursor * cursor);
```

For example:

```
samples_ctx->close_cursor(cur);
```

6.3 Move

Once opened, a cursor can be moved to a specific key. A cursor is initially positioned just before the key specified during its opening.

```
chr_cursor::move(const unsigned char * key, chr_pos_type type,
    chr_pos_where where);
```

- `type` can be:
  - `CHR_KEY` to open a cursor on a specific key,
  - `CHR_FIRST` to open a cursor on the first key of the table (`key` should be `NULL`),
  - `CHR_LAST` to open a cursor on the last key of the table (`key` should be `NULL`).
- `where` can be:
  - `CHR_BEFORE` to move the cursor just before the key,
  - `CHR_AFTER` to move the cursor just after the key,
  - `CHR_ON` to move the cursor on the key.

For example, to position `cur` just before the first key of the table:

```
cur->move(NULL, CHR_FIRST, CHR_BEFORE);
```

6.4 Insertion

To maximize performances, keys subsequently inserted with the same cursor should be increasing. Otherwise, the cursor performs a `move` transparently before the insertion. In both cases, the cursor is moved just after the inserted key.

Note: chronos does not allow duplicate keys within a table.

```
chr_cursor::insert(const unsigned char * key,
    const unsigned char * value);
```

For example:

```
unsigned char * key = (unsigned char *) malloc(sizeof(uint32_t) +
    sizeof(uint64_t));
unsigned char * value = (unsigned char *) malloc(sizeof(double));
SENSOR_ID(key) = htobe32(10);
TIMESTAMP(key) = htobe64(1224598516123);
VALUE(value) = 3.14;
cur->insert(key, value);
```

6.5 Deletion

To delete a key-value pair, the key must be specified.

```
chr_cursor::del(const unsigned char * key);
```

For example:

```
cur->del(key);
```

To delete a key-value pair with a specific value, the pair first has to be read (cf. section 6.7) and verified by the application.

6.6 Update

Chronos can update a value associated with a key. The key remains the same, only the value is overwritten.

```
chr_cursor::update(const unsigned char * key,
    const unsigned char * value);
```

For example:

```
cur->update(key, value);
```

To update a key, the application has to (optionally read and) delete the original key-value pair, and then insert the updated key-value pair.

6.7 Read

A cursor can be used to retrieve data with increasing keys.

```
chr_cursor::read_next(unsigned char * key, unsigned char * value);
```

For example, to read the content of the table `samples`:

```
cur->move(NULL, CHR_FIRST, CHR_BEFORE);
while (cur->read_next(key, value) != CHR_EOF)
{
  uint32_t sample_id = be32toh(SENSOR_ID(key));
  uint64_t timestamp = be64toh(TIMESTAMP(key));
  double sample = VALUE(value);
}
```

7 Return codes

Possible return codes for each method are defined in their header in the source code. To summarize, the following error codes are essential:

- `CHR_OK` no error,
- `CHR_ERR` table not available (`create_table`, `drop_table`, `close_table`),
- `CHR_EOF` end of table reached (`read_next`),
- `CHR_NOTFOUND` table not found (`drop_table`), or key not found (`del`, `update`),
- `CHR_DUPKEY` insertion of duplicate keys (`insert`).

References

[Chardin, 2011] Chardin, B. (2011). *Open-source DBMS for data historization and impact of flash memories*. Ph.D Thesis in Computer Science, INSA de Lyon.