Using libRaptorQ library

December 16, 2015

Abstract

libRaptorQ is a C++11 implementation of the RaptorQ Forward Error Correction, as described in the RFC6330 .

The implementation was started as a university laboratory project, and will be later used and included in Fenrir, the maintainer's master thesis.

This implementation is quite short (the core is $\sim 3k$ lines), thanks to the chosen language and the use of external libraries for matrix handling (eigen3).

libRaptorQ is the only RaptorQ implementation in C++, include C hooks, and it is the only free (**LGPL3**) implementation of the rfc, except for the (apache2) java implementation, OpenRQ , which is much bigger ($\sim 46k$) and slower.

Contents

1 Contacts **3**

2 Build & install **3**

2.1 Get the source code 3

2.2 Dependencies 4

2.3 Build & Install 4

3 Working with RaptorQ **6**

3.1 Theory (you really need this) 6

3.2 Blocks & Symbols 6

3.3 C++ interface 7

3.3.1 The Encoder 8

3.3.2 Blocks 10

3.3.3 Symbols 10

3.3.4 The Decoder 11

3.4 C interface 13

3.4.1 Common functions for (de/en)coding 14

3.4.2 Encoding 15

3.4.3 Decoding 16

4 GNU Free Documentation License **17**

1 Contacts

The main development and dicussions on the project, along with bug reporting, happens on the main website.

Mailing Lists Mailing lists are available at https://www.fenrirproject.org/lists The two mailing lists are for development and announcments, but due to the low traffic of the development mailing list, it can also be used by users for questions on the project.

IRC Since there are not many developers for now, the main irc channel is **#fenrirproject** on freenode

2 Build & install

2.1 Get the source code

Although things seems to work, no stable release has been released yet, as of December 16, 2015.

This means you can only check this out with git.

To check out the repository:

```
$ git clone https://github.com/LucaFulchir/libRaptorQ.git
```

You can also get it from our main server:

```
$ git clone https://www.fenrirproject.org/Luker/libRaptorQ.git
```

GPG verification: Once you have cloned it, it's always a good thing to check the repository gpg signatures, so you can import my key with:

```
$ gpg --keyserver pgp.mit.edu --recv-key D42DDF0A
```

please check the full fingerprint, it should be this:

```
$ gpg2 --fingerprint D42DDF0A
 pub   rsa3072/D42DDF0A 2015-01-01 [expires: 2016-01-01]
       Key fingerprint = AB35 E45F 5CA5 E35B 8B55  818F 0157 D133 D42D DF0A
 uid       [ unknown] Luca Fulchir (2015 key) <luker@fenrirproject.org>
```

Now you have the source, and the key, it's enough to check the signature of the last commit:

```
$ git log -n 1 --show-signature
```

The important part is that you get something like this:

```
 gpg: Signature made Fri 27 Mar 2015 20:59:59 CET using RSA key ID D42DDF0A
 gpg: Good signature from "Luca Fulchir (2015 key) <luker@fenrirproject.org>"
 [unknown]
 gpg: WARNING: This key is not certified with a trusted signature!
 gpg:          There is no indication that the signature belongs to the owner.
 Primary key fingerprint: AB35 E45F 5CA5 E35B 8B55  818F 0157 D133 D42D DF0A
 Author: Luca Fulchir <luker@fenrirproject.org>
```

And as long as you got the right key, and you find the **"gpg: Good signature"** string, you can be sure you have the right code.

2.2 Dependencies

libRaptorQ has only 2 dependencies:

Eigen3 : This is used for matrix manipulation, which is a big part of RaptorQ.

git : This is used not only to get the source, but also by the build system. We get the last git commit id and feed it to clang or gcc as seed for their internal random number generator. This makes it possible to have reproducible builds.

2.3 Build & Install

libRaptorQ uses the cMake build system, so things are fairly standard:

```
$ cd libRaptorQ.git
$ mkdir build
$ cmake ../
$ make -j 4
```

By default, the libRaptorQ project tries to have deterministic builds. This means that if you compile things twice, or with two different computers, the hash of the resulting library will be the same, provided that the same compiler (clang, gcc 4.8, gcc 4.9 etc) was used. Currently the only exception is the clang compiler with the *PROFILING* option enabled, and this will not likely be solved.

There are lots of options, you can use in cmake. As always, you can change them by adding "**-Dcmake_option=cmake_value**" when calling cmake.

You can always use the cmake-gui or ccmake commands to have the list of possible options.

The ones we recognize are:

LTO : yes/no. Default:**yes**. Enable *Link Time Optimizatios* for clang or gcc. Makes libraries smaller and better optimized.

PROFILING : yes/no. Default:**yes**. *WARN: breaks deterministic builds with clang.* Profiling compiles everything once, then runs a test to see which code paths are used more, and then recompiles everything again, but making sure that the binary is optimized for those paths. Works with clang and gcc. Provides a slight speedup.

CMAKE_C_COMPILER : gcc and clang are directly supported. other should work, too. This is only used if you want to build the C example.

CMAKE_CXX_COMPILER : g++, clang++ are directly supported. other should work, too.

STDLIB :	change the c++ standard library. "libstdc++" for the (default) gcc one, "libc++" for the clang/llvm one. Note that it seems you can't use libc++ with gcc yet.
CMAKE_CXX_FLAGS :	Additional compiler flags you might want to pass.
CMAKE_BUILD_TYPE :	Type of build. you can choose between "Debug", "Release", "MinSizeRel" or "RelWithDebInfo"
CMAKE_INSTALL_PREFIX :	Default to */usr/local*. Change it to fit your distribution guidelines.

Then you can build everything by running:

```
$ make -j 4
```

Of course, you can configure the number of parallel jobs (the *-j* parameter) to be what you need.

Optional make targets: The following optional targets are also supported:

```
$ make docs tests examples
$ make everything
```

The "docs" target builds this document, but you need latex with the refman style. The tests are only useful to check perfromance of rfc compliance right now. "examples" compiles the C and C++ examples, which will not be installed.

Install: The installation process is very simple:

```
$ make install DESTDIR=...
```

You can change the *DESTDIR* parameter to fit your distribution guidelines.

3 Working with RaptorQ

3.1 Theory (you really need this)

To be able to work with liRaptorQ, you must first understand how the RaptorQ algorithms works. We won't go into the details, but just what you need to be able to tune the algorithm to your needs.

Fountain codes: Fountain codes are a special *Forward-Error-Correcting* code class, which characteristic is simple: if you want to send K packets, you actually send $K + X$ packets, and the receiver only needs to get any K packets to be able to reconstruct your data. The number X of overhead packets can be as big as you need (theoretically infinite), so you can tune it to be slightly higher than the expected packet loss.

Systematic codes: RaptorQ is also a systematic code. This means that those first K packets are the input *as-is* (**source symbols**), and the X packets (**repair symbols**) have the information needed to recover any of the lost source packets. This has the big advantage of avoiding any kind of time and memory consuming decoding if there is no packet loss during the transmission.

Complexity: The RaptorQ algorithm is often presented as having a linear time encoder and decoder. This is both false and misleading. Generating the source or repair symbols from the intermediate symbols has linear complexity. Generating the intermediate symbols has cubic complexity on the number of symbols. Which is a completely different thing. It is still very quick. On a core i7, 2.4Ghz, you need to wait *0.4ms* for 10 symbols, *280ms* for 1.000 symbols, but it can take an hour for 27.000 symbols. RaptorQ handles up to 56.403 symbols.

3.2 Blocks & Symbols

To understand how to properly use and tune libRaptorQ, you first need to understand how RaptorQ handles its inputs, outputs, and what the time and memory constraints are.

Input sequencing: RaptorQ needs to have the whole input you want to send before it can start working.
This means that it might be a little more difficult to use in live-streaming contexts, or where you need real-time data, but libRaptorQ will have options to facilitate usage even in those contexts.

Once you have the whole input, RaptorQ divides it into **blocks**. Each block *is encoded and decoded independently* and will be divided into **symbols**. Each symbol *should* be transmitted separately in its own packet (if you are on the network).

Sizes: Each input can have up to *256 blocks*, each block can have up to 56.403 *symbols*, and each symbol can have up to $2^{16} - 1$ *bytes*long.

This gives a maximum files size of almost 881 GB (946270874880 bytes to be exact)

Interleaving: An other feature of RaptorQ is to automatically provide some interleaving of the input data before transmitting it. This means that one symbol will not represent one sequential chunk of your input data, but more likely it's the concatenation of different **subsymbols**. The size of the subsymbol must thus be a fraction of the symbol size. This feature is not used if you set the size of the subsymbol to the size of symbol.

Memory and Time: Memory and time requirements are to be considered, though, as RaptorQ needs to run a cubic algorithm on matrix of size $K * K$, where K is the number of symbols in each block.
The algorithm needs to keep in memory two of these matrices, although most of the work is done on only one.
This is actually a lot. More benchmarks and optimizations will come later, for now remember that with 10 symbols it takes something like 0.4ms on a core i7 2.4GHZ, 280ms with 1000 symbols, and up to an hour with 27.000 symbols.

3.3 C++ interface

To use the C++ interface you only need to include the **RaptorQ.hpp** header, and link **libRaptorQ** and your threading library (usually *libpthread*).

To provide grater flexibility, the whole library uses iterators to read your data, and to write the data onto your data structures.
This means that a big part of the library is a template, which adapts to the alignment of the data in the data structures you use.

Templates There are two main classes you will use:

```
template <typename Rnd_It, typename Fwd_It>
class Encoder

template <typename In_It, typename Fwd_It>
class Decoder
```

As you might guess, the classes for the encoder and decoder take two template parameters.
For the **Encoder**, the first parameter *MUST* be a *random access iterator* to the input data, and the second parameter is an *forward iterator*. The random access iterator will be used to scan your input data, and perform an interleaving (if you did not set the same size the symbol and to the subsymbol). The forward iterator will be used to write the data to your structure.
The same is done for the **Decoder**, but we do not need to do any interleaving on the input, so the first iterator can be just an input iterator, and nothing more.

3.3.1 The Encoder

You can instantiate an encoder for example by doing:

```
std::vector<uint32_t> input, output;
...
using T_it = typename std::vector<uint32_t>::iterator;
RaptorQ::Encoder<T_it, T_it> enc (input.begin(),
                                  input.end(),
                                  4, 1444, 10000)
```

This will create an Encoder that works on vectors of unsigned 32 bit integers for both input and output, that will create symbols of size 1444 bytes, interleaving your input every 4 bytes, and try to work with big number of symbols per blocks (**TODO: explain memory requirements**)

The available methods for the encoder are the following:

operator bool() : **return:bool**
False if constructor parameters did not make sense. Else true.

OTI_Common() : **return: OTI_Common_Data**, aka *uint64_t*.
Keeps total **file size and symbol size**. You need to send this to the receiver, so that it will be able to properly decode the data.

OTI_Scheme_Specific_Data() : **return: OTI_Scheme_Specific_Data**, aka *uint32_t*.
Keeps number of **source blocks, sub blocks, and alignment**. As for the OTI_Common_Data, you need to send this to the receiver to be able to properly decode the data.

encode : **Input: Fwd_It &output, const Fwd_It end, const uint32_t esi, const uint8_t sbn**.
return:uint64_t.
Take as input the iterators to the data structure into where we have to save the encoded data, the **Encoding Symbol Id** and the **Source Block Number**. As you are writing in C++, you probably want to use the iterators begin/end, though. Returns the number of written iterators (**NOT** the bytes)

encode : **Input: Fwd_It &output, const Fwd_It end, const uint32_t id**.
return:uint64_t.
Exactly as before, but the **id** contains both the *source block number* and the *encoding symbol id*

begin() : **return: Block_Iterator¡Rnd_It, Fwd_It¿**
This returns an iterator to the blocks in which RaptorQ divided the input data. See later to understand how to use it.

end() : **return: const Block_Iterator¡Rnd_It, Fwd_It¿**
This returns an iterator to the end of the blocks in which RaptorQ divided the input data. See later to understand how to use it.

precompute : **Input:const uint8_t threads, const bool background**
return: void
Do the work of computing all different blocks in multithread.
If *background* is true, then return immediately, else return only when the job is done.
If *threads* is 0, try to guess the maximum threads from the number of available cpus.

precompute_max_memory : **return: size_t**
Each precomputation can take a lot of memory, depending on the configuration, so you might want to limit the number of precomputations run in parallel depending on the memory used. This method returns the amount of memory taken by **ONE** precomputation.

free : **Input: const uint8_t sbn)**
return: void
Each block takes some memory, (a bit more than $symbols * symbol_size$), so once you are done sending source and repair symbols for one block, you might want to free the memory of that block.

blocks() : **return: uint8_t** The number of blocks.

block_size() : **Input: const uint8_t sbn**
return: uint32_t
The block size, in bytes. Each block can have different symbols and thus different size.

symbol_size() : **return: uint16_t** The size of a symbol.

symbols : **Input:uint8_t sbn**
return: uint16_t
The number of symbols in a specific block. different blocks can have different symbols.

max_repair : **Input: const uint8_t sbn)**
return: uint32_t
The maximum amount of repair symbols that you can generate. Something less than 2^{24}, but the exact number depends on the number of symbols in a block

3.3.2 Blocks

With the *begin()/end()* calls you get *Input iterators* to the blocks. a Block is has the following type:

```
template <typename Rnd_It, typename Fwd_It>
class Block
```

and exposes the 4 following methods:

begin_source : **return: Symbol_Iterator**

end_source : **return: Symbol_Iterator**

begin_repair : **return: Symbol_Iterator**

end_repair : **Input: const uint32_t max_repair**
return: Symbol_Iterator

max_repair : **return:uint32_t**

symbols : **return: uint16_t**

block_size : **return: uint32_t**

As the names explain, you will get an iterator to the symbols in the block. As the number of repair symbols can vary, for now you get two separate begin/ends, so that you can check when you sent the source symbols, and how many repair symbols you send.
The other functions are helpers for the details of the block.

3.3.3 Symbols

Finally, through the *Symbol_Iterator Input Iterator* we get the **Symbol** class:

```
template <typename Rnd_It, typename Fwd_It>
class Symbol
```

which exposes the 2 methods we need to get the symbol data:

operator* : **Input:Fwd_It &start, const Fwd_It end**
return: uint64_t
takes a forward iterator, and fill it with the symbol data. returns the number of written iterators.

id() : **return: uint32_t**
return the id ($sbn + esi$) of this symbol, that you need to include in every packet you send, before the symbols.

3.3.4 The Decoder

The decoder is a bit simpler than the encoder.

Theere are two constructors for the Decoder:

```
std::vector<uint32_t> input, output;
using T_it = typename std::vector<uint32_t >::iterator;
RaptorQ::Decoder<T_it, T_it> dec (
                const OTI_Common_Data common,
                const OTI_Scheme_Specific_Data scheme)

RaptorQ::Decoder<T_it, T_it> dec (uint64_t size,
                                  uint16_t symbol_size,
                                  uint16_t sub_blocks,
                                  uint8_t blocks)
```

Which should be pretty self-explanatory, once you understand how the encoder works.

The remaining methods are:

decode : **Input: Fwd_It &start, const Fwd_It end**
return:uint64_t
Write **all** the blocks into the iterator. refuses to write if the input has not been completely received. Return the number of iterators written.

decode : **Input: Fwd_It &start, const Fwd_It end, const uint8_t sbn**
return:uint64_t
Write a specific block into the iterator. Refuses to write if the input for that block has not been completely received.

add_symbol : **In_It &start, const In_It end, const uint32_t esi, const uint8_t sbn**
return: bool
Add one symbol, while explicitly specifying the symbol id and the block id.

add_symbol : **In_It &start, const In_It end, const uint32_t id**
return: bool
Same as before, but extract the block id and the symbol id from the *id* parameter

free : **Input: const uint8_t sbn**
return: void
You might have stopped using a block, but the memory is still there. free it.

blocks() : **return: uint8_t** The number of blocks.

block_size() : **Input: const uint8_t sbn**
return: uint32_t

The block size, in bytes. Each block can have different symbols and thus different size.

symbol_size() : **return: uint16_t** The size of a symbol.

symbols : **Input:uint8_t sbn**
return: uint16_t
The number of symbols in a specific block. different blocks can have different symbols.

3.4 C interface

The C interface looks a lot like the C++ one.
You need to include the **cRaptorQ.h** header, and link the *libRaptorQ* library.

Static linking

If you are working with the static version of libRaptorQ remember to link the C++ standard library used when compiling the library (*libstdc++* for gcc or maybe *libc++* for clang), your threding library (usually *libpthread*), and the C math library (*libm*).

First of all, you need to build the encoder or the decoder.

The C interface is just a wrapper around the C++ code, so you still have to specify the same things as before. A quick glance at the constructors should give you all the information you need:

C Constructors

```
typedef enum { NONE = 0,
        ENC_8 = 1, ENC_16 = 2, ENC_32 = 3, ENC_64 = 4,
        DEC_8 = 5, DEC_16 = 6, DEC_32 = 7, DEC_64 = 8}
             RaptorQ_type;

struct RAPTORQ_LOCAL RaptorQ_ptr
{
  void *ptr = nullptr;
  const RaptorQ_type type;
};

RaptorQ_ptr* RaptorQ_Enc (const RaptorQ_type type,
                    void *data,
                    const uint64_t size,
                    const uint16_t min_subsymbol_size,
                    const uint16_t symbol_size,
                    const size_t max_memory);

RaptorQ_ptr* RaptorQ_Dec (const RaptorQ_type type,
             const RaptorQ_OTI_Common_Data common,
             const RaptorQ_OTI_Scheme_Specific_Data
                                          scheme);
```

The encoder and decoder must have a specific alignment. in C++ you can also have different alignments for the input and output, while in C things are a bit more strict as we have to enumerate all the possible cases. So you only get the same data alignment for both input and output.
Still, you don't lose anything in performance.

3.4.1 Common functions for (de/en)coding

These functions are used by both the decoder and the decoder, and will be helpful in tracking how much memory you will need to allocate, or in general in managing the encoder and decoder.

The **ptr** must be a valid encoder or decoder. The names for now are self-explanatory. Blocks can have different symbols, so the size of a block and the number of symbols in a block depend on which block we are talking about.

symbols, blocks, memory

```
uint16_t RaptorQ_symbol_size (struct RaptorQ_ptr *ptr);
uint8_t RaptorQ_blocks (struct RaptorQ_ptr *ptr);
uint32_t RaptorQ_block_size (struct RaptorQ_ptr *ptr,
                                        const uint8_t sbn);
uint16_t RaptorQ_symbols (struct RaptorQ_ptr *ptr,
                                        const uint8_t sbn);

void RaptorQ_free (struct RaptorQ_ptr **ptr);
void RaptorQ_free_block (struct RaptorQ_ptr *ptr,
                                        const uint8_t sbn);
```

Finally, when you are done working with a block, you can free the memory associated with the single block and just free the whole (en/de)coder when you are done. Freeing the whole (en/de)coder will obviously free also all the blocks.

3.4.2 Encoding

OTI Data First, we need to tell the receiver all the parameters that the encoder is using, and for that two functions are provided:

```
typedef uint64_t RaptorQ_OTI_Common_Data;
typedef uint32_t RaptorQ_OTI_Scheme_Specific_Data;

RaptorQ_OTI_Common_Data RaptorQ_OTI_Common (
                                struct RaptorQ_ptr *enc);
RaptorQ_OTI_Scheme_Specific_Data RaptorQ_OTI_Scheme (
                                struct RaptorQ_ptr *enc);
```

Encoding

```
// maximum number of repair symbol in a block
uint32_t RaptorQ_max_repair (RaptorQ_ptr *enc,
                                    const uint8_t sbn);
//estimate bytes of ram used in the precomputation
// of one block
size_t RaptorQ_precompute_max_memory (
                                struct RaptorQ_ptr *enc);
// do the precomputation.
void RaptorQ_precompute (struct RaptorQ_ptr *enc,
                                  const uint8_t threads,
                                  const bool background);

// encode one symbol. source block number and
// symbol id are in the ``id'' field.
// returns number of alignments written.
uint64_t RaptorQ_encode_id (struct RaptorQ_ptr *enc,
                                   void **data,
                                   const uint64_t size,
                                   const uint32_t id);
// encode one symbol. same as before
uint64_t RaptorQ_encode (struct RaptorQ_ptr *enc,
                                   void **data,
                                   const uint64_t size,
                                   const uint32_t esi,
                                   const uint8_t sbn);
// build an ``id'' field out of an esi and sbn field.
uint32_t RaptorQ_id (const uint32_t esi,
                                   const uint8_t sbn);
```

As for the C++ version, everything is thread-safe.

You can start the precomputation in background and not worry about it.
If you request repair symbols before the computation is finished, the call will block until the data is available.

The **encode** functions are the same, and will encode **one** symbol. They work for both source symbols and repair symbols, just keep increasing the *esi* field

3.4.3 Decoding

Decoding

```
// return the total size of the data that will be
// decoded
uint64_t RaptorQ_bytes (struct RaptorQ_ptr *dec);

// decode all blocks.
// returns the number of written alignments.
// returns 0 if decoding of *everything* is not possible
uint64_t RaptorQ_decode (struct RaptorQ_ptr *dec,
                                    void **data,
                                    const size_t size);
// decode only one block (if possible)
// returns 0 if decoding of the block is not possible.
uint64_t RaptorQ_decode_block (struct RaptorQ_ptr *dec,
                                    void **data,
                                    const size_t size,
                                    const uint8_t sbn);

// add a received symbol to the structure.
// either by using the 'id' field
bool RaptorQ_add_symbol_id (struct RaptorQ_ptr *dec,
                                    void **data,
                                    const uint32_t size,
                                    const uint32_t id);
// or by explicitely declaring esi and sbn
bool RaptorQ_add_symbol (struct RaptorQ_ptr *dec,
                                    void **data,
                                    const uint32_t size,
                                    const uint32_t esi,
                                    const uint8_t sbn);
```

4 GNU Free Documentation License

GNU Free Documentation License
Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the

processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover

Index

Blocks
    C++, 10

cMake, 4
Contacts, 3

Decoder
    C, 16
    C++, 11
dependencies, 4

Encoder
    C, 15
    C++, 8

GPG, 3

Install, 5
Interface
    C, 13
    C++, 7
Interleaving, 7
Iterators, 7

Sequencing, 6
    Sizes, 6
Static Linking, 13
Symbols
    C++, 10

Theory, 6