librcm — reusable C modules

Frank Braun

2020-08-18

Table of Contents

This document describes how to write *reusable C modules* and documents `librcm`.

Check https://librcm.org for updates. Source code on GitHub.

Introduction

The C programming language was first introduced in 1972 and first standardized in 1989 as ANSI C (called C89). The most important revision of ANSI C in regards to this document is C99. We ignore all later revisions.

The importance of C today

According to the TIOBE Index for August 2020 C is the most popular programming language in the world.

Many operating systems, programming languages, and widely used libraries are written in C.

High-quality examples:

- The OpenBSD operating system.
- The Lua programming language.
- The SQLite database engine.

Many higher-level programming programming languages allow to interface with the C ABI.

Several new programming languages have the explicit goal to produce libraries compatible with the C ABI, for example Zig and Rust. Others, like Go, are criticized for not conforming to the C ABI. Conforming to the C ABI is mutually exclusive with garbage collection.

The importance of C in the future

Given the importance of C today, it's high performance, it's high portability (if written correctly), and the rise of low-performance IoT devices, C is likely to stay for many decades.

This means that **well written** C can be used and reused for a very long time. However, C suffers from a lot of poor quality code. The C ecosystem also lacks a repository of **easily reusable** C libraries, especially ones that can be embedded in other libraries, can be used in embedded devices, and are highly portable.

The problem of poor quality code is being addressed by restrictive coding standards and static analysis tools.

Reusable C libraries, however, are still a major problem. This is mostly due to the fact that the C programming language is so low-level and flexible that it neither defines a standard way on *how* to write libraries, nor does it have a standardized way to handle errors, manage memory, and deal with dependencies.

Having standardized ways to do things greatly helps to reduce cognitive overhead thereby and raises program quality and programmer productivity.

This is arguable a big reason that Go became so successful. The language itself is not extremely powerful, but making package names part of the language specification and having widely shared

opinions in the Go community what comprises idiomatic code makes Go highly reusable and reduces the cognitive burden when reading and editing code written by others.

In this document we describe a method on how to write highly quality, future-proof C code, giving rationales for decisions made. The power of this approach mainly comes from the **simplicity** that comes from standardizing on the *only true-way of doing things* ™. It forces the programmer in standardized way of writing C code with less decisions to be made along the way, freeing up mental capacity for writing the *actual* code.

But it comes at the cost that even less C code is reusable, because it doesn't conform to the future-proof C method. However, as more and more C code becomes a future-proof **reusable C module** this is less and less of a problem.

Resusable C modules

What does module mean?

A C module is a single C header file which defines and implements an API that can be used by *consumers* of the module. The module should *consume* as little other modules and header files as possible, to keep the dependency graph small.

What does reusable mean?

Reusable means that such a module should be useful on as many platforms as possible. Supported platforms include:

- All major POSIX systems: Linux, Mac OS, OpenBSD, and FreeBSD.
- Both major mobile platforms: Android and iOS.
- Embedded systems (like Arduino/AVR microcontrollers).
- 32-bit and 64-bit systems.
- Little-endian and big-endian systems.
- All major compilers: GCC, Clang, MinGW, and Visual Studio.
- All major processor architectures: x86, ARM, and PowerPC.

Goals

- Modules must not leak symbols into the global name space.
- Modules must be compilable as `static` in an amalgamation file, to make them reusable in libraries without any symbol leakage.
- Modules must minimize cognitive overhead. Standardize everything.
- Modules should avoid dynamic memory allocation wherever possible.

See also the STB how-to.

Error handling

TODO:

- error codes
- error messages
- error flow
- exceptions

Memory managment

TODO

How to write a module

A reusable C module `mod` consists of four components:

1. The header file: `rcm_mod.h`
2. The implementation C file: `rcm_mod.c`
3. The unit test file: `rcm_mod_test.c`
4. The module documentation (generated): `rcm_mod.adoc`

Only the header and the C files are really necessary to reuse the module. The online documentation for a module is linked from the header file.

We write the documentation in a separate file for the following reasons:

- It gives us the design capabilities of a lightweight markdown language.
- Generated documentation is often of poor quality, because it is written as an afterthought. Writing the documentation separately, in the API design stage, makes it better.

How to write the header file

```
/* SPDX-License-Identifier: Unlicense OR MIT */ ①

#ifndef RCM_MOD_H ②
#define RCM_MOD_H

/* Documentation: https://librcm.org/#rcm_mod.h */ ③

#ifndef RCM_API
#define RCM_API extern ④
#endif

/*
Example module: ⑤

Example module which shows how to write _reusable C modules_. ⑥
*/

/* Does nothing. */ ⑦
RCM_API void rcm_mod_foo(void); ⑧

/* Returns 23. */
RCM_API int rcm_mod_bar(void);

#endif /* RCM_MOD_H */ ⑨
```

① License identifier

② Include guard

③ Link to online documentation

④ Define `RCM_API`. Allows to compile all module functions as `static`

⑤ Module name, ends with `:`

⑥ Module description

⑦ Function comment(s)

⑧ Function definition(s)

⑨ End of include guard

Rules

- Write portable ANSI C89.
- `#include <stdbool.h>` is OK.
- Modules are single header files.
- Public domain for new code (use the [Unlicense]).
- Permissive license (BSD, MIT, or ISC) for modified third party code.
- No GPL.
- Format the code with `clang-format` (standardized `.clang-format` file).
- Write unit tests for the code with the greatest testing library.

Allowed defines

The number of defines used in reusable C modules must be kept to the absolute minimum! Otherwise `#ifdef`-hell ensues.

List of allowed `#ifdef`-statements:

- `ANDROID`: Android.
- `APPLE`: Apple platform (MacOS *and* iOS).
- `linux`: Linux (also defined on Android).
- `FreeBSD`: FreeBSD.
- `OpenBSD`: OpenBSD.
- `_MSC_VER`: The Visual Studio compiler is used. Defined as an integer literal that encodes the major and minor number elements of the compiler's version number.
- `NDEBUG`: combined with `#ifndef` to disable debugging code.
- `TARGET_OS_IOS`: Is `== 0` for Mac OS and `> 0` for iOS (on Apple platform).
- `_WIN32`: Windows (usually with `#ifndef` to implement POSIX first).

Coding style

- Error buffer: `char err[ERRBUF_SIZE]` (module `rcm_errbuf.h`)
- Return variables: `int ret`

Example module

TODO: `rcm_mod.h` example module.

How to test

TODO:

- valgrind, clang, greatest
- assertions
- allowed types
- licensensing

Use ANSI C89

Stick to C89 with the following exceptions from C99:

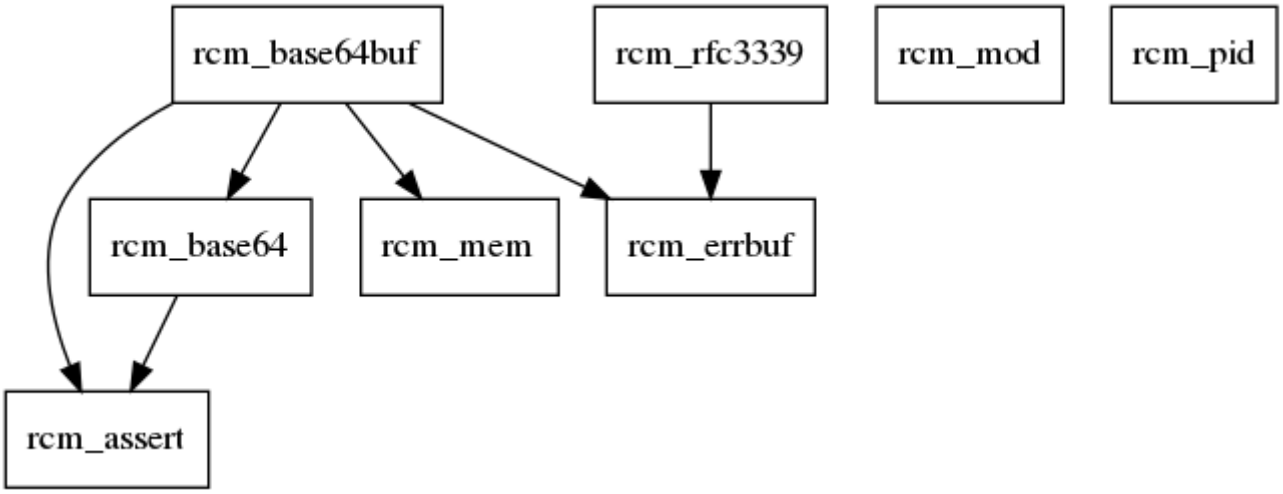
- `#include <stdbool.h>` is allowed.

Rationale

Visual Studio has poor support for C99.

`goto` is not allowed

`librcm` module dependencies



librcm module APIs

rcm_assert.h — Robust assertions

The robust assertion macro is inspired by the "JPL Institutional Coding Standard for the C Programming Language", Rule 16, (see https://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf).

Recommended use:

```
if (!rcm_assert(p >= 0)) {
  return RCM_ASSERT_ERR_FAILED;
}
```

Compile with -DNDEBUG for production and without for development and testing.

`#define rcm_assert(e) rcm_assert_handle((e), FILE, LINE, #e)`

Robust assertion macro.

`void rcm_assert_set_handler(rcm_assert_handler_func *handler);`

Set the assertion handler. The default is rcm_assert_default_handler. Set to NULL to disable logging of assertions.

`void rcm_assert_set_handler_data(void *data);`

Set data pointer passed to the assertion handler.

`bool rcm_assert_default_handler(const char *file, int line, const char *expr, void *data);`

The default assert handler. Prints the failed assertion to stderr and then calls abort(3) if not compiled with NDEBUG.

`bool rcm_assert_handle(bool assert_result, const char *file, int line, const char *expr);`

Internal function. It calls the rcm_assert_handler, if not NULL and assert_result is false.

rcm_base64.h — Base64 encoding

This module implements low-level funtions for the base64 standard encoding with padding as described in RFC 4648. Use the rcm_base64_buf module for higher-level functions (with memory allocation and error buffers).

`size_t rcm_base64_encode_len(size_t len);`

Returns the base64 encoded length of a binary input buffer of size len.

`rcm_base64_err_t rcm_base64_encode(char *out, const unsigned char *in, size_t len);`

Writes the base64 encoding of the input buffer in of lenght len to out and appends a `\0` character. The output buffer out has to be at least the size `rcm_base64_encode_len(len)+1`!

`size_t rcm_base64_decode_len(size_t len);`

Returns the decoded length of an base64 encoded input buffer of size len.

`rcm_base64_err_t rcm_base64_decode(unsigned char *out, const char *in, size_t len);`

Decodes the base64 encoded input in of length len and writes it to out (if not NULL). If the the output buffer out is not NULL it has to be at least of the size `rcm_base64_decode_len(len)`!

`const char *rcm_base64_errstr(rcm_base64_err_t errnum);`

Return an error string describing the given error number errnum. For RCM_BASE64_OK an empty string is returned ("", not NULL).

`rcm_base64buf.h` — Base64 buffer

This module implements high-level functions for the base64 standard encoding with padding as described in RFC 4648. Use the rcm_base64 module for lower-level functions (without memory allocation and error buffers).

`rcm_base64buf_err_t rcm_base64buf_encode(char **out, const unsigned char *in, size_t len, char *err);`

Encodes the base64 encoding of the input buffer in of lenght len and writes it it as a null-terminated string to out.

`rcm_base64buf_err_t rcm_base64buf_decode(unsigned char **out, size_t *outlen, const char *in, size_t len, char *err);`

Decodes the base64 encoded input in of length len and writes it to the freshly allocated out buffer. If outlen is not NULL, the length of the output buffer is written to it. In case of error out remains unmodified, an error code is returned, and an error message is written to err.

`const char *rcm_base64buf_errstr(rcm_base64buf_err_t errnum);`

Return an error string describing the given error number errnum. For RCM_BASE64BUF_OK an empty string is returned ("", not NULL).

`rcm_errbuf.h` — Error buffer

The error buffer module.

`#define RCM_ERRBUF_SIZE 1024`

The default error buffer size.

`void rcm_errbuf_set(char *err, const char *format, ...);`

Sets error buffer err to format string (printf(3)-like) with arguments ⋯. Does nothing, if err is `NULL`.

`rcm_mem.h` — Memory allocator

This modules defines the memory allocator for *reusable C modules*.

`void *rcm_mem_malloc(size_t size);`

Allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized.

`void rcm_mem_free(void *ptr);`

Frees the memory space pointed to by ptr.

`rcm_mod.h` — Example module

Example module which shows how to write *reusable C modules*.

`void rcm_mod_foo(void);`

Does nothing.

`int rcm_mod_bar(void);`

Returns 23.

`rcm_pid.h` — Process ID

Module to deal with process IDs (PIDs).

`int rcm_pid_get(void);`

Returns the process ID (PID) of the calling process.

`rcm_rfc3339.h` — RFC3339 time

Module to deal with time strings in RFC3339 format.

`#define RCM_RFC3339_BUFSIZE 21`

The default RFC3339 string buffer size.

`typedef struct rcm_rfc3339_t rcm_rfc3339_t;`

The RCF3339 time type.

`int rcm_rfc3339_parse(rcm_rfc3339_t *time, const char *value, char *err);`

Parse time `value` in RFC3339 format and store the result in `time` (if not `NULL`). Returns -1, if the parsing failed. 0, otherwise. In case of error, the error message is written to `err`.

`int rcm_rfc3339_now(rcm_rfc3339_t *t, char *err);`

Return the current time (now) in UTC and store it in `t`. Returns -1, if the time could not be determined. 0, otherwise. In case of error, the error message is written to `err`.

`void rcm_rfc3339_format(char *out, rcm_rfc3339_t time);`

Format time as RFC3339 in UTC to `out` buffer (`\0`-terminated). The `out` buffer must have at least size `RCM_RFC3339_BUFSIZE`!

`bool rcm_rfc3339_after(rcm_rfc3339_t ltime, rcm_rfc3339_t rtime);`

Return true if `ltime` is after `rtime`, false otherwise.

`bool rcm_rfc3339_before(rcm_rfc3339_t ltime, rcm_rfc3339_t rtime);`

Return true if `ltime` is before `rtime`, false otherwise.

`rcm_rfc3339_t rcm_rfc3339_from_time_t(time_t time);`

Convert from `time_t` to `rfc3339_t`.

Deliverables

- user guide: How to write future-proof C. With coding rules and rationales.
- reference: What APIs are offered?
- tutorials: How to write library module. How to use library.