librcm — reusable C modules

Frank Braun

2020-08-17

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.

Error handling

TODO:

- error codes
- error messages
- error flow
- exceptions

Memory managment

Rules

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 APIs

`rcm_errbuf.h` — Error buffer

`RCM_ERRBUF_SIZE`

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_rfc3339.h` — RFC3339 time

`RCM_RFC3339_BUFSIZE`

The RCM3339 buffer size.

`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. Returns -1, if the parsing failed. 0, otherwise. In case of error, the error message is written to `err`.

`rcm_rfc3339_t rcm_rfc3339_now(void);`

Return the current time (now) in UTC.

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

Format time as RFC3339 in UTC to out buffer (`\0`-terminated). Out 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.